AD-A054 009 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 17/2
MULTIPLE-PROCESSOR IMPLEMENTATIONS OF MESSAGE-PASSING SYSTEMS.(U)
APR 78 R H HALSTEAD N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-198 NL

1 OF 2
AD A
054009

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

12 sc

MIT/LCS/TR-198

MULTIPLE-PROCESSOR IMPLEMENTATIONS OF MESSAGE-PASSING SYSTEMS

Robert H. Halstead, Jr.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

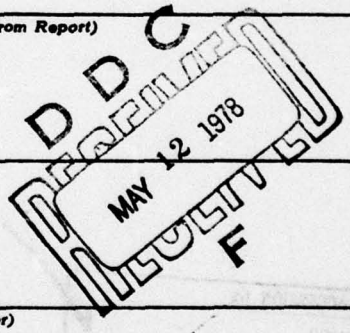
SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MIT/LCS/TR-198	2. GOVT ACCESSION NO. (9) Master's thesis,
4. TITLE (and Subtitle) (6) Multiple-Processor Implementations of Message-Passing Systems	5. TYPE OF REPORT & PERIOD COVERED: S.M.Thesis, Jan. 20, 1978
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-198
7. AUTHOR(s) (10) Robert H. Halstead, Jr.	8. CONTRACT OR GRANT NUMBER(s) (15) N000014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT/Laboratory for Computer Science, 545 Technology Square, Cambridge, Ma 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, Department of Defense, 1400 Wilson Boulevard, Arlington, Va 22209	12. REPORT DATE (11) Apr 78
	13. NUMBER OF PAGES (12) 174 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Va 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)



18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Message passing
distributed computing
actor semantics
networks

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

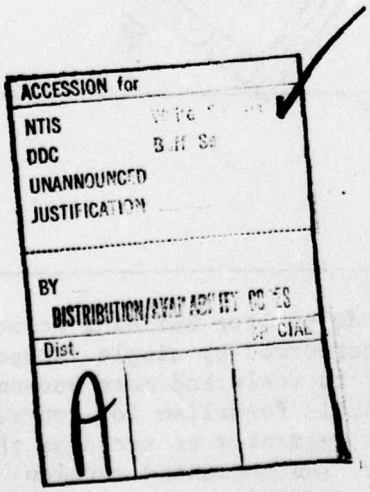
The goal of this thesis is to develop a methodology for building networks of small computers capable of the same tasks now performed by single larger computers. Such networks promise to be both easier to scale and more economical in many instances. The mu calculus, a simple syntactic formalism for representing message-passing computations, is presented and augmented to serve as the semantic basis for programs running on the network. The augmented version includes cells, tokens, and semaphores, as well as primitives for side-effect-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

409 648

20. free computation. Tokens, a novel construct, allow certain simple communication and synchronization tasks without involving fully general side effects. The network implementation presented supports object references, keeping track of them by using a new concept, the reference tree. A reference tree is a group of neighboring processors in the network that share knowledge of a common object. Also discussed are mechanisms for handling side effects on objects and strategy issues involved in allocating computations to processors.



MIT/LCS/TR-198

MULTIPLE-PROCESSOR IMPLEMENTATIONS OF MESSAGE-PASSING SYSTEMS

by

Robert Hunter Halstead, Jr.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

CAMBRIDGE MASSACHUSETTS 02139

MULTIPLE-PROCESSOR IMPLEMENTATIONS OF MESSAGE-PASSING SYSTEMS

by

Robert Hunter Halstead, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 1978 in partial fulfillment of the requirements for
the Degree of Master of Science

ABSTRACT

The goal of this thesis is to develop a methodology for building networks of small computers capable of the same tasks now performed by single larger computers. Such networks promise to be both easier to scale and more economical in many instances.

The mu calculus, a simple syntactic formalism for representing message-passing computations, is presented and augmented to serve as the semantic basis for programs running on the network. The augmented version includes cells, tokens, and semaphores, as well as primitives for side-effect-free computation. Tokens, a novel construct, allow certain simple communications and synchronization tasks without involving fully general side effects.

The network implementation presented supports object references, keeping track of them by using a new concept, the reference tree. A reference tree is a group of neighboring processors in the network that share knowledge of a common object. Also discussed are mechanisms for handling side effects on objects and strategy issues involved in allocating computations to processors.

Name and Title of Thesis Supervisor:

Stephen A. Ward
Assistant Professor of Electrical Engineering and Computer Science

Key Words and Phrases:

Message-passing, distributed computing, actor semantics, networks.

ACKNOWLEDGMENTS

Primary credit for the accomplishments, though not blame for the faults, of this thesis must go to my thesis supervisor, Steve Ward, who supplied crucial inspiration and managed to keep the thesis on track while at the same time convincing me that it was I who was keeping it on track.

Secondary credit must be shared by the whole DSSR group at MIT, for upgrading and maintaining the UNIX timesharing system on which the thesis research was performed, and on which the thesis document itself was prepared. John Pershing, Tom Teixeira, Terry Hayes, and other individuals mentioned in this introduction all contributed software that was useful to me. I am grateful to all who uncomplainingly tolerated my simulation runs, many of which decimated system performance.

Important intellectual contributions arose out of many discussions with Jim Gula, Chris Terman, Peter Jessel, and others. Jim Gula in particular probably deserves credit for steering my research in this direction in the first place.

Clark Baker deserves special mention for his careful reading of Chapter 2, which uncovered several errors of varying severity.

The Department of Electrical Engineering and Computer Science at MIT through their teaching support and the United States Government through their research support have been responsible for keeping a roof over my head thus far during my career as graduate student and thesis writer.

Finally, I must express my gratitude to my parents, not only for their support over the years, but for the use of their quiet home, where much of the most productive work on this thesis was actually performed.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

TABLE OF CONTENTS

Chapter 1: Introduction

The goal of this thesis is to develop a methodology for building networks of small computers capable of the same tasks now performed by single larger computers. An important reason for preferring a properly designed network over a single processor is the ease of scaling the network configuration up or down by simply changing the number of processors, rather than undergoing the trauma of, say, switching to a more powerful central processor. Moreover, at one extreme of this scaling approach lies the possibility of constructing networks with capacity exceeding that of any feasible single processor.

Another primary motivation for using networks instead of single processors is economic. A collection of slower processors with smaller memories is likely to be less expensive than a single processor with the same aggregate instruction execution rate and the same total memory size. This economic argument is valid, however, only if the collection of smaller processors can be used as effectively as the large processor to solve the problems of interest. In practice, there is likely to be some overhead incurred in distributing the computation. Fortunately, the economic advantages of distribution are strong enough to outweigh a modest penalty in this department.

1.1: Parallelism

Several broad categories of research are relevant to this thesis. One such category is the study of parallel algorithms and hardware configurations. The inclusion of parallelism in an algorithm is significant because it relaxes the constraints on how that algorithm must be executed (specifically, it relaxes constraints on the ordering of steps in the algorithm). This relaxation of constraints allows for new

possible implementations. In particular, it may make feasible the use of special parallel hardware configurations. Some of these, like Illiac IV[3], are designed for a fairly narrow range of applications (numerical array processing, in this case). Others, such as proposed designs for data flow machines[27] aim to be more generally usable.

The purpose of most of these machines is to speed the completion of a computation by doing several parts of it in parallel. This reduction in total elapsed time for a computation is also an important goal of this thesis, but there are differences in approach. First, the method to be presented in this thesis is generally suited to hardware of relatively traditional construction (e.g., microprocessors). Second, much of the parallelism research cited above has focused on very high-performance machines designed to compute faster than any single computer could, whereas our approach aims also to be feasible at lower points on the spectrum. Third, special-purpose parallel designs tend to restrict the kinds of computations the programmer can specify if he wants to be able to use the full power of the system. While our system (or, for that matter, even a single-processor system!) cannot be totally immune from this sort of effect, an effort has been made not to exclude any particular kind of programming tool (e.g., side effects) from consideration. Although some computations may run faster than others, our system will do its best to apply the maximum possible parallelism to all problems.

1.2: Networks and Distributed Computing

Another category of research related to this thesis deals with computer networks. The prototypical large computer network is the ARPANET[21]; however, for our purposes we shall be more interested in some of the newer small-computer networks, such as Ethernets[22], ring networks[8], and shared buses[23,36]. Research involving these networks is aimed at increasing the reliability and flexibility of computer systems, as well as simply facilitating communication between existing computers. In most cases, unfortunately, useful results in this area have been limited to hardware technology. No particularly successful software principles have emerged for using these networks.

Such work as has been done on the software aspects of computer networks mostly falls under the rubric of "distributed computing." Work of this type has been conducted by several researchers, notably Farber[9,10,25,26] and Wulf[37]. Distributed computing inevitably entails some amount of parallelism, the difference being that in what is usually thought of as distributed computing, the coupling between parallel activities is looser. Often a distributed system is conceived of as simultaneously performing several *independent* tasks, almost like a timesharing system, whereas students of parallelism are more apt to concentrate on coordinated tasks devoted to solving a single problem.

The "timesharing system" aspect of distributed computing raises new questions that transcend those simply involving parallelism. A distributed system must face issues of coherence, security, reliability, and capacity for growth. In many cases, these issues arise in distributed systems in a form substantially different from their form in centralized computer systems. The topics of security and reliability have been largely ignored in this thesis; however, coherence and capacity for growth

are very important goals of the system design we present.

In spite of these common interests, the research described here is fundamentally different from most distributed-systems research. The latter has been concerned mainly with ways of interconnecting existing systems, or systems resembling them, with minimum disruption. Such work is of great practical importance, but the research in this thesis is intended to be more visionary. Thus there is no explicit attempt to deal with existing systems. Instead, a new kind of system is designed from the ground up (or perhaps from the sky down, as we shall see!).

Finally, the development of protocols for packet communication (e.g., on the ARPANET) is another subject in the distributed-computing area with some relevance to this thesis. This topic is not central to the thesis, but research on protocols will be cited from time to time in support of the feasibility of the communication algorithms to be presented.

1.3: Thesis Overview

Most distributed-computing research to date has either consisted of hardware designers coming up with neat new gadgets, or software designers trying to figure out what to do with them. The philosophy of our research is that the emphasis on hardware technology in network design has been misplaced. A top-down approach is needed: first determine the software capabilities and organizational principles that are desirable for a network, then pick a suitable technology and build the network.

This thesis documents an attempt to design a network in just this manner. It begins with a description of a simple language based on the semantics of message passing, and proceeds to show a multiprocessor implementation of this language

and explore its properties.

Although we shall attempt to justify each choice, the course of research such as this leads past many arbitrary decision points. No claim can be made that a message-passing language is the only reasonable starting point. Similarly, many features of the various implementations presented are arbitrary; no "optimal" implementation has been developed. The message-passing language itself is simple in the extreme. Although it contains all the essential elements to construct a practical system, any language to be used in real life would obviously be considerably richer in features. Work of this scope only scratches the surface of what is possible in multiprocessor systems.

Finally, the reader should be alerted to the fact that the implementation described in this thesis has only been tested in the most preliminary way. The skeleton of the system design has been shown to function; that is all that can be said with certainty. Much work remains to be done before it can be determined whether the approach developed here can really serve as the basis for an efficient and useful system.

1.3.1: Message-Passing Semantics

Chapter 2 of the thesis addresses the first step in our top-down design process: the choice of a semantic basis for the programming languages we might imagine supporting on our system. Almost immediately, we must choose between two approaches: fairly standard process-based semantics, and the newer message-passing outlook.

In the process model, the state of a computation is thought of, in general, as composed of the contents of state variables (program counter, environment pointer) associated with each of the various *processes* that currently exist in the system.

In this model, communication and synchronization between parallel processes is frequently a problem, and the emphasis on sequentiality within each process (even though other processes may run in parallel) combines with the real or imagined overhead of creating and destroying processes to discourage the expression of parallelism possible on the microscopic ("statement" or "expression") level. Thus the process model encourages or forces the programmer to make arbitrary decisions about microscopic sequencing of computations. Once the decisions have been made, it can be hard to rediscover the valid alternatives.

At first glance, the process model might seem well suited for use on a network of processors--simply divide up the processes among the processors available, relying on a scheduler in each processor to arrange for the sharing of that processor among the processes assigned to it. The problem with this idea is that, using existing formalisms, the programmer is not likely to create very many processes which can all be active at the same time. This, in turn, is likely to lead to a succession of local imbalances, with the bulk of the processing load descending in turn on one small set of processes (and processors) after another. In fact, the current state of the art in multiprocessor systems seems largely to consist of distributing processes among processors *by hand* in such a fashion as to minimize imbalances.

In response to this situation, Chapter 2 presents a language and semantic structure which encourage the expression of microscopic parallelism, which limit in a natural way the amount of storage directly accessible at any particular point in execution (so that a particular locus of control can be economically moved between processors), and which help solve the problems of communication and synchronization between co-operating activities. This semantic structure will be based on the message-passing model of computation, exemplified by the actor systems of Hewitt[14,15,16] and Alan Kay's SMALLTALK[19]. This approach to computing

appears to be fundamentally different from the "applicative" model, typified by the lambda calculus of Alonzo Church and the programming language LISP, and the "imperative" model, embodied in the Von Neumann machine and most traditional programming languages. Although Hewitt's PLASMA, for instance, contains numerous constructs of an applicative or imperative flavor, it differs from conventional approaches in that the underlying semantics of all these constructs are specified exclusively in terms of message passing.

In a message-passing system, an *event* is defined as the receipt of a message by an *actor*. When an actor receives a message, its *script* is invoked, which may in turn cause other events. This simple message-passing control structure can replace procedures, iteration, sequencing, and so on. The entire state of a computation at any point may be summed up by the set of events that remain to be processed, the system *event list*. Of course, this is only true if we regard the processing of an event as an atomic, indivisible operation; when an event is removed from the event list, any events it might cause to be added to the event list will appear in the same quantum of time. Whenever we look, we see the system "at rest," and its entire future potential is a function only of the contents of the event list.

The "mu calculus" of Ward and Halstead[35] is the starting point in our development of a semantic foundation, and forms the basis for what we shall call "pure" message-passing systems with no "side effects." In these systems, not only have the traditional control structures of function application, iteration, etc., been dispensed with, even the notion of "process" is no longer required.

Pure message-passing systems are a useful introductory framework for studying various aspects of message passing, but are too restrictive to serve as models for many important situations. Consequently, Chapter 2 will be concerned with

extending the message-passing language so that it contains the semantic constructs important for most programming tasks. This will be done by introducing two new kinds of objects: *cells* and *tokens*. Cells provide the basic mechanism for side effects.

Tokens, which resemble the tokens described by Henderson[13], can be used for many of the same purposes as cells, but avoid some of the problems of implementing cells on distributed systems. A token is like a "pipe" for communication between events that are separated in time or space. Initially, the pipe is empty, but if an object is fed into one end of the pipe, it will be communicated to any objects that are present at the other end. Although they cannot be used to implement cells, tokens *can* be used to create self-referential structures and solve some simple synchronization and communication problems.

Chapter 2 concludes with a look at primitives for mutual exclusion. Throughout the chapter, examples show the relationship between this message-passing language and more traditional applicative and imperative languages. The semantics of the message-passing language itself are described in as abstract and formal a manner as feasible, with relatively little reference to possible implementations. The chapter is intended to be complete in itself, for the convenience of those readers whose main interest is in understanding the message-passing language and its extensions.

The purpose of developing the mu calculus is twofold. On the one hand, the mu calculus is designed to serve as the semantic basis or "machine language" for a distributed system. Actual programming would almost certainly be done in a highly sugared version of the mu calculus or in some language with a completely different appearance. Consequently, the mu calculus itself need not be a masterpiece of human engineering; rather, it should be (and is) a simple but representative

embodiment of the various operations that a distributed system must be capable of. The second, and related, purpose of introducing the mu calculus is as an attempt to capture the essence of message-passing computation in a simple formalism. The conclusion of Chapter 2 includes comments relating to the success of this endeavor.

1.3.2: Implementations

Chapter 3 of the thesis describes an implementation of the message-passing language on a network of processors. The physical structure chosen is one in which each processor has a limited number of neighbors (e.g., four) with which it can communicate directly. Communication between processors that are not immediate neighbors must be handled by one or more intermediary processors. The system supports object references[4] and uses an event list distributed among the various processors to keep track of the state of pending computations. A system standard external representation for data is assumed, though representations inside processors may differ. Garbage collection and mutable objects are supported. Some attention is paid to the problem of scheduling events for maximally efficient operation.

The main contribution of this chapter is in the object management algorithms discussed in Section 3.3, particularly the reference tree concept and the management of mutable objects. The very important strategy issues treated in Section 3.4, the reader is once again advised, must still be regarded as open questions. Satisfactory performance of the whole system must await better solutions to these problems than those actually tried thus far.

Since it attempts to give a complete description of the implementation, the narrative in Chapter 3 is occasionally encumbered by a level of detail greater than

some readers will want. Such readers may skim or skip Sections 3.3.4.1 and 3.4.1 with no loss of continuity.

Chapter 3 relies only on the most fundamental of the concepts set forth in Chapter 2; readers primarily interested in implementations can consider themselves qualified to move on to Chapter 3 as soon as they feel they have a very basic understanding of actors, events, tokens, and cells. In particular, comprehension of the translation rules or examples given in Chapter 2 is not required.

1.3.3: Conclusion

Chapter 4 of this thesis presents conclusions and suggestions for further research. It reviews the thesis's attempt at a top-down system design effort, discusses briefly some alternatives to the design given here, and points out some directions for continued work on the mu calculus and distributed system design.

Chapter 2: Message-Passing Semantics

The goal of this chapter is to develop a semantic structure able to make effective use of a suitably organized network of processors. We begin with an extremely simple message-passing language.

2.1: The Pure Mu Calculus

The pure mu calculus is a basic formalism for representing message-passing computations. It is "pure" in that it contains no mechanism for causing side effects; thus all "objects" representable in the calculus are immutable. Such a language, akin to the lambda calculus (often used as a simple model of applicative languages), lends itself easily to formal treatment with fairly well-established techniques. In fact, as we shall see, the mu calculus is like a restriction of the lambda calculus in which certain kinds of expressions are not allowed. The price we pay for this ease of formal treatment is that the mu calculus is too simple a language to provide a satisfactory model for many interesting situations. Nevertheless, we study the pure mu calculus before going on because it exposes the basic philosophy and character of message-passing systems, as well as providing a framework for subsequent embellishments. It should be added that the ease of formal description of the pure mu calculus is accompanied by a great deal of flexibility in how to implement a distributed system based on this calculus. In fact, one result of this thesis is to document a certain correlation between formal tractability and flexibility of implementation. This correlation appears to be much stronger on distributed systems than in centralized systems. For example, side effects, which are difficult to handle formally, cause no particular implementation problems on centralized computer systems, but can cause quite a bit more difficulty on a network.

Thus the pure mu calculus, although not very useful by itself, forms a distinctly useful subset of a fuller message-passing language, especially one designed for a distributed system.

2.1.1: Formal Definition of the Mu Calculus

The introduction stated that the basic elements of a message-passing system are events, actors, and scripts. While the distinction between an actor and a script will be useful to us later, for now we will just represent an actor by writing its script, and will use the concepts interchangeably. Other kinds of objects, such as numbers, are also useful, so following [35] we make the following definitions:

Definition 2.1:

An *object* is

1) a member of a set C of *distinguished constants* (such as numbers),
2) a member of a set V of *variables* (for our purposes we shall represent specific variables by upper- and lower-case Roman letters),
3) a member of a finite set P of *primitives* (primitive operations such as addition, comparison, and so on), or
4) an *actor* of the form $(\mu x_1 x_2 \cdots x_n . E_1 E_2 \cdots E_m)$ where each of the x_i is a variable and each of the E_i is an *event*. For any $n \geq 0$ and $m \geq 0$ such an expression is a valid actor.

Definition 2.2:

An *event* is a sequence $(A_1 A_2 \cdots A_n)$ of objects, for any $n \geq 1$.

In many cases, parentheses in events or actors are redundant and may be supplied from context; in such cases, they may be omitted.

The above definitions establish objects and events as two of the fundamental constituents of the mu calculus. The third major ingredient, which makes the mu calculus a dynamic system rather than just a static definition of objects and events, is the *causality* relation, which relates pairs of events. It will be useful in what follows to distinguish pairs of events and objects that differ in some structural way from pairs that differ only in the names chosen for bound variables.

Definition 2.3:

Two events or objects *A* and *B* are *congruent*, written $A \sim B$, if *A* can be converted to *B* simply by renaming bound variables in *A*. For this purpose we use the same definition of "bound variable" as is used in the lambda calculus[5,6].

Note that congruence is an equivalence relation. For the remainder of this thesis, we shall treat congruent events or objects as if they were the same. Thus, formally, we are working with *equivalence classes* of events or objects, and representing these classes by selected members.

We may now discuss the causality relation --> on events. The causality relation is defined by two axioms which closely parallel the axioms of the lambda calculus. For the first axiom, which corresponds to beta-reduction in the lambda calculus, we must have a syntactic substitution function like the lambda-calculus substitution rule $S[X;y;Z]$ which denotes, informally, the result of substituting the expression *X* for all free occurrences of the variable *y* in the expression *Z*, renaming bound variables in *Z* if necessary to avoid identifier conflict. The mu-calculus substitution function is identical to this in every respect except the replacement of the letter λ by the letter μ, so interested readers are referred to [6] for the

details. In defining the mu calculus, it is frequently useful to be able to express the result of substituting several expressions for several variables at the same time. We write this as $S[X_1,X_2,\cdots,X_n;y_1,y_2,\cdots,y_n;Z]$. This is equivalent to

$$S[X_1;y_1;S[X_2;y_2;\cdots S[X_n;y_n;Z]\cdots]]$$

provided that none of the y_i occurs free in any of the X_j, which can always be arranged by renaming the offending y_i and similarly renaming all free occurrences of y_i in Z.

We are now prepared to state the axioms that define causality:

Axiom A1 ***(mu-reduction)*:**

If E is the event $((\mu x_1 x_2 \cdots x_n . E_1 E_2 \cdots E_m) A_1 A_2 \cdots A_n)$, then E --> E' for any event E' of the form $S[A_1,A_2,\cdots,A_n;x_1,x_2,\cdots,x_n;E_i]$ where $1 \le i \le m$.

Axiom A2 ***(primitives)*:**

If E is the event $(pc_1c_2\cdots c_nA)$ where p is a primitive and each c_i is a constant, then E --> $(A\ \tilde{p}[c_1;c_2;\cdots;c_n])$, where $\tilde{p}$ is the function denoted by p.

It is often convenient to use the transitive closure of the relation -->; following usual mathematical practice, we denote this by $-->^+$. Similarly, we denote the reflexive transitive closure of --> by $-->^*$.

2.1.2: Discussion

The pure mu calculus has been shown to be a consistent and universal computing scheme[35]. Let us examine in more detail the various elements of this scheme.

Events are the basic mechanism by which things "happen" in the mu calculus. An event $(A_1 A_2 \cdots A_n)$ denotes the arrival of a *message* containing the sequence of objects $A_2, \cdots, A_n$ at the receiver object A_1. Two kinds of objects are meaningful as receivers: actors (with meaning formally specified by axiom A1) and primitives (dealt with by axiom A2).

Objects represent the data of the mu calculus. Distinguished constants may be used to model numbers and other fundamental data types whose semantics need not be further specified within the axiom system of the mu calculus. Primitives stand for the basic operations on these constants. Identifiers are placeholders; when an actor which binds a particular occurrence of an identifier receives a message, the occurrence of the identifier is replaced by the corresponding object from the message. Finally, actors are the principal mechanism for abstraction in the mu calculus. The body of an actor may contain events which are prototypes for the computations that will ensue whenever the actor receives a message.

Events can *cause* other events; a computation in the mu calculus generally proceeds from some *initial event*, through various events caused by this event and its descendants, to some desired final event or events caused (in the sense of $-->^*$) by the initial one. Axioms A1 and A2 specify the mechanisms by which new events may be caused, and in so doing give meaning to actors and primitive objects.

Axiom A1 deals with actor-events (events whose receivers are actors) and, as

has already been mentioned, bears a close correspondence to axiom beta of the lambda calculus. The significance of axiom A1 is that if the actor $(\mu x_1 x_2 \cdots x_n . E_1 E_2 \cdots E_m)$ receives a message $A_1 A_2 \cdots A_n$, then m new events can be caused, one corresponding to each of the E_i in the body of the actor, with the appropriate object A_j substituted for each free occurrence in E_i of each bound variable x_j of the actor. For example:

(μxc.+xxc)3R causes +33R

(μc.(+33c)(c4))R causes +33R and R4

(μxc.)7R causes no event

Thus an actor with one event in its body will cause one new event when it receives a message, while an actor with more than one event in its body will cause several events. An actor with an empty body will terminate the particular line of computation from which it was sent a message. In this way, the pure mu calculus includes mechanism for spawning and terminating concurrent activities. It unfortunately does not, as we shall see, include any mechanism for communication between concurrent activities.

Axiom A2 provides for certain primitive functions which operate on the distinguished constants of the computing scheme. The exact nature of these functions is not important, and we will invent new ones wherever convenient. The only restriction is that a primitive function must have only sequences of constants in its domain. Examples of the kinds of functions that are useful are

*AC which causes (C A+1)

+ABC which causes (C A+B), and

>ABC which causes (C μabc.ca) if $A > B$, and causes (C μabc.cb) if $A \leq B$.

Thus

*8R causes R9

+34R causes R7

>67R causes Rμabc.cb

>52μx.x10R causes (μx.x10R)(μabc.ca)

2.2: Coding Applicative Constructs

Since message-passing is a relatively novel and unconventional approach to computing, it is appropriate to describe its relationship to more familiar approaches. Some literature on this topic has been written by Hewitt[15], but the work described here is even more radical than his, since he still allows some applicative constructs to be present in his message-passing language. This work is also carried out at a more fundamental level than Hewitt's, simplifying many of the analogies. Consequently, several sections in this chapter will consider translations between mu-calculus constructs and those of other programming methodologies. In this section, we focus on applicative languages, such as the lambda calculus or Pure LISP.

2.2.1: A Normal-Order Translation

A rule for translating applicative expressions into objects is stated and proved in [35]; we give it, without proof, below. Central to the translation is the concept of a *continuation*, as described by Strachey and Wadsworth[32] or Hewitt[15]. In an applicative language, the use to which the value of an expression will be put is determined by the context in which that expression appears. In the mu calculus, there is no such thing as an "expression." Values are computed and used by means of events causing other events. In the mu calculus, there is also no "context" (in the applicative sense) for a computation. Thus a value that is computed by, say, adding two numbers, must be disposed of in a manner specified by another part of the same event which caused the addition. The prevailing ethic for doing this kind of computation in a message-passing language is to use a continuation, an actor which will receive the value produced by a computation and which will then continue, using that computed value where appropriate.

The translation rule we present takes an applicative expression *A* and produces an object *O*[*A*] with the property that if *O*[*A*] is sent some continuation *C*, *C* will at some point be sent the value of the original applicative expression *A* (appropriately translated to the mu-calculus domain). Thus

$$O[3]\ C \dashrightarrow^{*} C3$$

$$O[+25]\ C \dashrightarrow^{*} C7$$

$$O[\lambda x.x]\ C \dashrightarrow^{*} C(\mu xc.cx)$$

Definition 2.4:

Given any applicative expression A, the object $O[A]$ is

1. If A is a constant n, then $\mu c.cn$.
2. If A is an identifier x, then x.
3. If A is a λ-expression $\lambda x_1 \cdots x_n.M$, then $\mu c.c(\mu x_1 \cdots x_n c.(O[M]\ c))$ where c is an identifier that does not occur free in M and is not on the list $x_1,\cdots,x_n$.
4. If A is a combination $PQ_1 \cdots Q_n$, then $\mu c.(O[P]\ \mu y.(y\ O[Q_1] \cdots O[Q_n]\ c))$, where c is an identifier that does not occur free in $Q_1,\cdots,Q_n$.
5. If A is a unary primitive (such as σ), then $\mu c.c(\mu ad.a(\mu x.px(\mu z.(\mu e.ez)d)))$, where p is the mu-calculus primitive corresponding to A.
6. if A is a binary primitive (such as + or >), then $\mu c.c(\mu abd.a(\mu x.b(\mu y.pxy(\mu z.(\mu e.ez)d))))$, where p is the mu-calculus primitive corresponding to A.

Translations for other primitive operators are similar to those specified in clauses 5 and 6.

As shown in [35], the existence of this translation rule establishes the universality and consistency of the mu calculus. Perhaps more importantly for the purposes of this work, it gives a clue as to how various useful lambda-calculus devices, such as the Y operator, may be adapted for use in the mu calculus.

The translation rule O models in the mu calculus the semantics of *normal-order reduction* on applicative expressions. An interesting attribute of the mu calculus is that it can be used to model other reduction orders as well (see [6] for a discussion of reduction orders). This is because the order in which events are caused in the mu calculus is highly constrained by the requirement that axioms can only be applied to entire events, not to subexpressions of events. Thus the next computa-

tion is the one that is on the top level of an event; subsequent computations are specified by subexpressions which are nested more deeply inside the event. This degree of explicitness can be both an asset and a liability, as we shall see.

2.2.2: An Applicative-Order Translation

A reduction order that is used more commonly than normal-order reduction is *applicative-order reduction*, in which the values of arguments are computed before the function to be applied to them is invoked. The use of applicative-order reduction usually simplifies an interpreter and increases its efficiency relative to normal-order reduction; this accounts for its use in LISP and similar languages. It is possible to describe translation rules from applicative expressions to message-passing expressions which produce results corresponding to applicative-order reduction on the applicative expression being translated.

One such possible rule is (for simplicity, we treat only single-argument functions)

Definition 2.5:

Given any applicative expression A, the object $O'[A]$ is

1. If A is a constant n, then $\mu c.cn$.
2. If A is an identifier x, then $\mu c.cx$, where c is some identifier other than x.
3. If A is a λ-expression $\lambda x.M$, then $\mu c.c(\mu xc.(O'[M]\ c))$ where c is some identifier other than x which does not occur free in M.
4. If A is a combination PQ, then $\mu c.(O'[P]\ \mu y.(O'[Q]\ \mu z.yzc))$, where c is an identifier that does not occur free in A and y is an identifier that does not occur free in Q.
5. If A is a unary primitive (such as σ), then $\mu c.cp$, where p is the mu-calculus primitive corresponding to A.

Comparison of this definition with definition 2.4 reveals that the evaluation of an argument expression X has been transferred from the application of primitives (clause 5) back to the evaluation of arbitrary combinations (clause 4). ("Evaluation" of X, in mu-calculus terms, means the causing of an event $(O[X]\ C)$ which will eventually cause C to be sent the value of X.) The change in evaluation times requires a corresponding change in the semantics of identifiers in the translation O'. In the processing of events generated by the translation rule O, identifiers are replaced by unevaluated arguments, that is, objects of the form $O[X]$ for some X. This is a consequence of the form of clauses 3 and 4 of definition 2.4. The change in clause 4 of definition 2.5 causes identifiers to be bound instead to *values*, that is, objects returned to the continuation C by events of the form $(O'[X]\ C)$. Thus if we mean to abide by our previously established convention that $(O'[A]\ C)$ returns to the continuation C the value of the applicative expression A, we must define $O'[a]$ as $\mu c.ca$ for any identifier a. Thus, for example, if a is

replaced by the value 3, $O'[a]$ will become $\mu c.c3$, which is consistent with our convention.

2.2.3: Other Reduction Orders

The translation rules presented above show (1) that any computation expressible in the lambda calculus is expressible in the mu calculus, and (2) that the mu calculus removes certain ambiguities about reduction order which are present in the lambda calculus. Thus it is possible to write different translation rules which effectively specify different reduction orders for the applicative expressions being translated. Exponents of multiprocessing[2], though, have pointed out ways of turning the ambiguity of the lambda calculus to advantage, using parallel reduction orders where several subexpressions of an applicative expression are evaluated simultaneously, perhaps by several processors operating in parallel. It is not possible to write translation rules into the pure mu calculus which exhibit this kind of behavior, since the pure mu calculus forces the explicit specification of the order of events. At first sight, the ability of actor bodies to contain multiple events might seem to provide the basic mechanism for this kind of parallel evaluation, but there is no "join" operator to co-ordinate the results from two parallel chains of events and use both to compute some final output. The next section describes an extension to the mu calculus, called *tokens*, which permits the construction of a "join" operator as well as providing certain other capabilities.

2.3: Tokens

A characteristic of the pure mu calculus is that it forms a purely additive axiom system—if we denote by E^* the complete set of events that could eventually be caused by a set E of initial events, then for two sets of initial events E_1 and E_2, $(E_1 \cup E_2)^* = E_1^* \cup E_2^*$ This characteristic makes it impossible to write any kind of "join" operator in the pure mu calculus, for such an operator would allow us to specify, for example, that some event E was to occur as a result of the occurrence of both E_1 and E_2, but not simply as a consequence of either occurring alone. Thus $E \in \{E_1,E_2\}^*$ but $E \notin \{E_1\}^*$ and $E \notin \{E_2\}^*$. This contradicts the above observation that $\{E_1,E_2\}^* = \{E_1\}^* \cup \{E_2\}^*$ and requires instead that $\{E_1\}^* \cup \{E_2\}^* \subseteq \{E_1,E_2\}^*$. The idea that a set of several initial events ought to be able to have consequences that would not be caused by any one of those events individually suggests that a new kind of mu-calculus axiom is needed. Axioms A1 and A2 showed how a single event could cause other events. We now see the need for axioms wherein the conjunction of some set of events may cause other events. This is what a "join" primitive would do, and is also a capability sufficient for certain other simple communication and synchronization tasks. This capability is provided by extending the pure mu calculus to include tokens.

Definition 2.6:

A *token* is an element of the set

$$T = \{\langle r_X, w_X \rangle \mid X \text{ is an object in the mu calculus}\}$$

For any particular token $\langle r_X, w_X \rangle$, r_X is known as the *read side* and w_X as the *write side* of the token.

Axiom A3 ***(tokens)*****:**

The pair of events $r_X A$ and $w_X B$ (where $\langle r_X, w_X \rangle \in T$) *together* cause the event *AB*.

Additionally, a mechanism must be provided for generating tokens.

Axiom A4 ***(creation of tokens)*****:**

An event of the form τC causes the event $Cr_C w_C$.

A useful way of visualizing a token is as a pair of tables as shown in Figure 2.7.

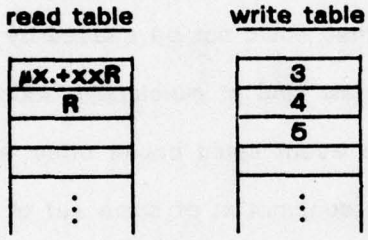


Figure 2.7: Viewing a token as tables.

Every time the read side of a token receives a message, the item received is entered in the first empty slot of that token's read table; messages received by the write side are similarly entered in the write table. At any point, the set of events that can have been caused by sending messages to the token is exactly the set of all events *AB* such that *A* appears in the read table and *B* appears in the write table.

Assuming that *X* is the identifier of the token depicted in Figure 2.7, the figure shows the state of affairs that would exist after the five events

$$r_X\ \mu x.+xxR$$

$$r_X\ R$$

$$w_X\ 3$$

$$w_X\ 4$$

$$w_X\ 5$$

had been caused, and would in turn lead to the eventual causation of the six events

$(\mu x.+xxR)3$	R3
$(\mu x.+xxR)4$	R4
$(\mu x.+xxR)5$	R5

A useful implementation of tokens would of course not wait for all operations on the token to have been completed before causing any of these events, but rather would generate the events incrementally. For example, every time an entry was added to the write table it could simultaneously be sent to each object currently present in the read table; addition of an entry to the read table could be handled similarly.

From this analogy it is possible to infer several properties of tokens:

1. If a token stops receiving messages, every value appearing in the write table (i.e., that was sent to the write side) will ultimately be sent to each object appearing in the read table (i.e., that was sent to the read side).

2. A token never loses information—a value, once received, remains entered in the appropriate table forever.

3. The (externally detectable) state of a token depends only on the identity of

the objects communicated to the two sides of the token, not on the order in which the messages were received (the position of an item in a read or write table is not significant, only its presence).

4. In their full generality, tokens form a rather bizarre control structure.

Properties (1) and (2) have simple explanations in terms of the axiomatic definition of tokens. Property (3) can be explained by noting that axiom A3 does not place any ordering on messages received by either side of a token; consequently, the ordering introduced in the "table" analogy is artificial and cannot be detected except by examining the table. These three properties indicate that tokens are a special kind of object, in a twilight zone between immutability and complete mutability. It is possible to modify a token by sending it a message, but such a modification can only add information to the token—old information can never be destroyed. Thus, for example, an old copy of a token is guaranteed to contain a subset of the information present in the current version of the token. Consequently, an old copy of a token never becomes invalid in the sense of containing incorrect information; it simply becomes less and less useful as additional information accumulates in the primary copy of the token. Tokens share this characteristic with, for example, eventcounts as described by Reed and Kanodia[24], where an old value of an eventcount is never dangerous; it may just be sufficiently out of date to be of little use for the intended application. Such methods of constraining the mutability of objects promise to be of considerable use in implementing distributed systems.

Property (4) raises two questions: what are tokens good for? and how can tokens be implemented practically? The first of these questions is addressed in the next section. As for the second, one possible implementation is suggested by

the "table" model. Further consideration and refinement should properly await the discussion in the next section showing the typical ways in which tokens might actually be used. In fact, discussion of these implementation details will be postponed until the next chapter.

2.4: Parallel Evaluation and Self-Reference

2.4.1: Parallel Evaluation of Applicative Expressions

We now return to the subject being discussed before the introduction of tokens, namely the translation of a program expressed in an applicative language such as the lambda calculus into an equivalent program expressed in the mu calculus. An objection that had been raised to the translation rules presented was that they specified particular evaluation orders and hence destroyed some information present in the original regarding flexibility of evaluation order. In particular, there are some possibilities for concurrency that do not violate the semantics of applicative expressions, but the pure mu calculus provides no way of expressing the possibility of such concurrency in the translation. This deficiency of the pure mu calculus was shown to be related to the impossibility of writing a "join" operator. We now show (by example) that it *is* possible to express this kind of parallel evaluation in the mu calculus with tokens.

We shall continue to restrict our attention to single-argument functions; hence, the kind of parallel evaluation we shall investigate is the evaluation of the operator of a combination in parallel with that of its operand. In order to co-ordinate these activities, a new token will be created every time the evaluation of a combination is begun. The read side of the token will be given to the operator as a "future" (see Baker and Hewitt[2]) for the value of the operand, and the write side will be held for the operand to send its value to. Thus the process of evaluation of the

operator will, whenever it requires the value of the operand, send a continuation to the read side of the token; if the operand value has been computed, the continuation will then be sent that value. Otherwise, that line of activity will cease until the operand value becomes available (i.e., is sent to the write side of the token), at which point all continuations sent to the read side of the token will receive the operand value. In detail, the new translation rule is as follows:

Definition 2.8:

Given any applicative expression *A*, the object $O''[A]$ is

1. If *A* is a constant *n*, then $\mu c.cn$.
2. If *A* is an identifier *x*, then *x*.
3. If *A* is a λ-expression $\lambda x.M$, then $\mu c.c(\mu xc.(O''[M]\ c))$ where *c* is some identifier other than *x* which does not occur free in *M*.
4. If *A* is a combination *PQ*, then $\mu c.(\tau(\mu rw.(O''[Q]\ w)(O''[P]\ \mu y.yrc)))$, where *c*, *r*, and *w* are identifiers that do not occur free in *A*.
5. If *A* is a unary primitive (such as σ), then $\mu c.c(\mu ad.a(\mu x.px(\mu z.(\mu e.ez)d)))$, where *p* is the mu-calculus primitive corresponding to *A*.

This translation rule closely resembles the normal-order translation rule *O* given in definition 2.4. In fact, other than the restriction to single-argument functions, the only difference is in the handling of combinations in clause 4. In this translation rule O'', as described above, whenever a combination is to be evaluated a new token is created using the τ operator. The resulting read and write sides r and w replace the variables *r* and *w* respectively, and two parallel events are caused. The first initiates a chain of events culminating in the receipt by w of the value of the operand *Q*; the second leads to the continuation μy.yrc receiving the value of *P*, presumably some sort of function, after which execution of the body of the

function will begin, with the formal parameter replaced by the read side r.

It is interesting to note that any variable present in the original applicative expression *A* will appear in the translation *O"*[*A*] in such a manner that it will eventually be replaced by the read side of some token (unless the variable is never replaced at all). Thus all situations which under ordinary evaluation would call for a variable in *A* to be bound to a value will instead result in the variable being bound to a future for that value. In this sense, the translation *O"*[*A*] represents a maximally parallel evaluation of *A*: every evaluation begins as soon as it can, and no evaluation waits for another until a primitive requiring the value of some future is invoked.

An amusing property of this scheme is that if *A* has any operands with no normal form, then even though *O"*[*A*] may eventually cause the value of *A* to be returned to its continuation, some sub-evaluations spawned by *O"*[*A*] will never return any value to their continuations. In a straightforward implementation of the mu calculus, these evaluations will continue forever! Even if these evaluations terminate, but just take a long time, they are still bothersome. The problem of identifying and "garbage-collecting" these irrelevant computations is dealt with by Baker and Hewitt[2] who use a somewhat different concept of "future" than that described here, however. Fortunately, any expression whose applicative-order evaluation (as defined by definition 2.5) terminates is guaranteed not to leave any nonterminating evaluations under the rule *O"*. Unfortunately, restricting ourselves to this class of expressions rules out some interesting possibilities, such as parallel evaluation of the two arms of a conditional before it is determined which arm is desired.

2.4.2: Parallel Evaluation of Actors

So far, we have been devoting considerable attention to translation rules from applicative expressions to mu-calculus objects. This has not been in an attempt to demonstrate the feasibility of a system which performs this translation before evaluating an applicative expression (although that is one possible goal), but an attempt to show the fairly straightforward relationship that exists between applicative programming and message-passing programming. The aim has been to show that the message-passing model combines the ability to specify fixed orders of evaluation with (when augmented by tokens) the ability to specify many useful forms of concurrency. As another illustration of this property, we consider the construction of a mu-calculus parallelism actor π.

The behavior we desire is that an event πABC will ultimately cause the event CXY, where X is the value A returns to its continuation and Y is the value B returns to its continuation (i.e., for any R, $AR \dashrightarrow^* RX$ and $BR \dashrightarrow^* RY$). Thus if A and B contain two calculations which may be carried out in parallel, π arranges for them to happen concurrently, and after both have finished, for the resulting values to be forwarded to C. Using tokens, π may be implemented as

$$\pi \equiv \mu abc.\tau\mu rw.(aw)(b\mu y.r\mu x.cxy)$$

The idea is that the value X will be computed and sent to the write side of the token, while the computation of Y is occurring simultaneously. When Y is computed, it is sent to the continuation $\mu y.r\mu x.Cxy$ which leads to the event $r\mu x.CxY$, where r is the read side of the token. When both sides of the token have received their respective messages, the event $(\mu x.CxY)X$ will then be caused, leading in turn to the desired consequence, CXY.

π is an attractive package for this functionality, and might well be the user's

main interface with tokens in a programming language based on the mu calculus. As an example of what π might be good for, consider two functions f and g, and suppose we wanted to compute the sum $f(3)+g(4)$, evaluating the two terms in parallel. In mu-calculus parlance, this would become

$$\pi(\mu c.f3c)(\mu c.g4c)(\mu xy.+xyR)$$

Although π can be constructed using tokens, the author has discovered no way to construct a token using π and believes this task to be impossible. This accounts for the decision to axiomatize tokens rather than the more obvious but seemingly less general construct π.

2.4.3: Representing List Structures

In preparation for discussing another application of tokens, we digress briefly to consider how LISP-like list structures might be represented in the mu calculus. The fundamental LISP list-structure operator is the constructor function cons, which takes two items and binds them together into one. In message-passing terms,

$$(\text{cons } A\ B\ C) \dashrightarrow^{*} (C\ X)$$

where X contains within it the arguments A and B. We may write cons as

$$\text{cons} \equiv \mu abc.c(\mu x.xab)$$

in which case

$$(\text{cons } A\ B\ C) \dashrightarrow (C\ \mu x.xAB)$$

The constructor function cons is complemented by two selector functions car and cdr, expressed as

$$\mathbf{car} \equiv \mu xc.x(\mu ab.ca)$$

$$\mathbf{cdr} \equiv \mu xc.x(\mu ab.cb)$$

which select out the first and second components, respectively, of an item constructed by **cons**. Cast into message-passing form, the familiar identities regarding the relationship between **car**, **cdr**, and **cons** are

$$(\mathbf{cons}\ A\ B\ \mu x.(\mathbf{car}\ x\ C)) \dashrightarrow^* CA$$

$$(\mathbf{cons}\ A\ B\ \mu x.(\mathbf{cdr}\ x\ C)) \dashrightarrow^* CB$$

Verification that these identities are true of our implementation is left to the reader.

2.4.4: Self-Reference

A capability of tokens which is quite independent of their usefulness in parallel evaluation is their use to create self-referential structures. In the lambda calculus, a kind of self-reference can be achieved by using the Y operator

$$Y \equiv \lambda g.(\lambda h.g(hh))(\lambda h.g(hh))$$

Thus in the expression $Y(\lambda f.F)$, free occurrences of the identifier f in the expression F effectively refer to the expression F itself. This device is obviously available in the pure mu calculus by simply using the translation rule in definition 2.4.

Tokens provide a similar capability in a way which is somewhat less magical than the machinations of the Y operator. Using tokens is likely to be more efficient (depending on the implementation of tokens) and is certainly more straightforward. As a simple example, let us consider the manufacture of a circular list whose car contains some datum A and whose cdr is the list itself. In the pure mu calculus, an

object such as this would have to be constructed with the help of some actor derived from the Y operator. Using tokens, we can construct the list (and send it to our result continuation R) starting with the following event:

$$\tau(\mu rw.(\text{cons } A\ w\ (\mu x.(R\ x)(r\ x))))$$

The chain of events that will ensue is as follows (abbreviating by Z the token identifier $\mu rw.(\text{cons } A\ w\ (\mu x.(R\ x)(r\ x)))$):

$$(\text{cons } A\ w_Z\ (\mu x.(R\ x)(r_Z\ x)))$$
$$(\mu x.(R\ x)(r_Z\ x))(\mu x.x\ A\ w_Z)$$
$$(R\ (\mu x.x\ A\ w_Z))\ \text{and}\ (r_Z\ (\mu x.(x\ A\ w_Z)))$$

Let us use B to denote the actor $\mu x.xAw_Z$ received by R. Then (car B C) will eventually cause (C A), as we would hope, and (cdr B C) will cause (C w_Z). Let us see what happens if this value w_Z is ever passed to one of the list-structure selectors, say car.

$$(\text{car } w_Z\ C)$$
$$\text{--> } w_Z\ (\mu ab.Ca)$$
$$\text{--> } (\mu x.xAw_Z)(\mu ab.Ca)$$
$$\text{--> } (\mu ab.Ca)Aw_Z$$
$$\text{--> } CA$$

The only tricky part of the above sequence occurs between the second and third lines, where the message sent to w_Z is sent in turn to the actor $\mu x.xAw_Z$. This happens because that was the (only) value sent to r_Z when the self-referential structure was created. If the operator in this example had been cdr instead of car, then the last line above would have been Cw_Z instead of CA. Thus we really

have managed to capture the essence of a list structure which loops back on itself.

Unfortunately, this example only happens to work so nicely because the object returned by cons is a single-argument actor. In any other case, the current token mechanism, which only allows the sending of single-element messages through the pipe, would have been inadequate. This is an indication of a kind of inflexibility of tokens that we shall discuss again after seeing how tokens can be used to implement recursion.

2.4.5: Recursion

Throughout this section, we will assume that the recursive function of interest is defined as a recursive kernel F in which free occurrences of the identifier f represent recursive references to F. An example of such a definition for the factorial function would be

$$F \equiv \mu nc.>n1\mu t.(t(\mu c.-n1\mu m.fm\mu x.*nxc)(\mu c.c1)(\mu a.ac))$$

From F we can construct a "functional" F^* in which f is bound:

$$F^* \equiv \mu fc.cF$$

Now, using an approach similar to that pursued in the last section, we can write the event

$$\tau(\mu rw.(F^*(\mu xc.r\mu g.gxc)(\mu h.(Rh)(wh))))$$

which sends to R an object F which is the recursive function intended by F (i.e., the least fixed point of the functional F^*). Abstracting out the functional F^* and the continuation R, we can derive a kind of mu-calculus Y operator Y_μ using tokens

as

$$Y_\mu \equiv \mu fc.r(\mu rw.(f(\mu xc.r\mu g.gxc)(\mu h.(ch)(wh))))$$

A formal statement of the abovementioned fixed-point property is that for any continuation C and functional F^*,

$$(Y_\mu\ F^*\ C) \text{ -->}^*\ CF$$

where F is the least fixed point of F^*.

As mentioned in the previous section, recursion is possible even in the pure mu calculus, by translation from lambda-calculus recursion if by no other route. For comparison, then, here is an equivalent operator Y_μ which does not use tokens:

$$Y_\mu \equiv \mu fc.\Delta\Delta c \quad \text{where} \quad \Delta \equiv \mu gc.f(\mu xc.gg\mu h.hxc)c$$

Unlike the lambda-calculus Y operator, however, which can compute the least fixed point of an arbitrary expression, these operators Y_μ depend on *F* being a single-argument function (strictly speaking, on *F* being an actor that accepts two-element messages). Thus perhaps each of these should really have been labelled $Y_{\mu 1}$ to emphasize this restriction. It is of course possible to construct a $Y_{\mu 2}$; the version using tokens is

$$Y_{\mu 2} \equiv \mu fc.r(\mu rw.(f(\mu xyc.r\mu g.gxyc)(\mu h.(ch)(wh))))$$

Nevertheless, it is annoying to have to have all these different versions of Y. One solution to this problem is to change the semantics of the original recursive kernel *F* so that free occurrences of *f* refer not to *F* itself but to some object (such as the read side of a token) which will return *F* to its continuation. Using these modified semantics, the recursive kernel for the factorial function would become

$$F \equiv \mu nc.>n1\mu t.(t(\mu c.-n1\mu m.f\mu g.gm\mu x.*nc)(\mu c.c1)(\mu a.ac))$$

The functional F^n is, as before, defined to be

$$F^n \equiv \mu fc.cF$$

Now we may define a generalized mu-calculus fixed-point operator Y_μ (using tokens) as

$$Y_\mu \equiv \mu fc.\tau(\mu rw.(fr(\mu h.(ch)(wh))))$$

The equivalent operator in the pure mu calculus is

$$Y_\mu \equiv \mu fc.\Delta\Delta c \quad \text{where} \quad \Delta \equiv \mu gc.f(\mu c.ggc)c$$

2.4.6: Conclusions Regarding the Use of Tokens

One conclusion to be drawn from the above discussion of self-reference and recursion is that tokens might be more useful if it were possible to send other than single-element messages from the write side to the read side of a token. For example, we might consider changing axiom A3 to

Axiom A3' ***(extended tokens)*****:**

The pair of events $r_X A$ and $w_X B_1 B_2 \cdots B_n$ (where X is the same object in both cases) *together* cause the event $AB_1B_2\cdots B_n$.

With this new axiom, the use of tokens for recursion would become simpler and the objections raised in connection with generating self-referential list structures (that it was only by accident that tokens turned out to be directly applicable) would be answered. However, there is no increase in power as a result of this change. For the token described in axiom A3' to be used productively, any object A sent to the

read side of the token had better be of the appropriate functionality to receive an *n*-element message. Similarly, all messages received by the write side had better have *n* elements, otherwise they will not fit the actors waiting on the read side. Given that *n* is thus effectively fixed for any particular token, we can use our knowledge of *n* to simulate a new-style token using an old-style token as follows:

$$r'_X \equiv \mu a.r_X \mu v.va$$

$$w'_X \equiv \mu b_1 b_2 \cdots b_n.w_X \mu a.ab_1 b_2 \cdots b_n$$

where r'_X and w'_X are the sides of the new "token," and r_X and w_X are the sides of a token as described in axiom A3. The principal advantage of defining tokens as in axiom A3' is that the value of *n* need not be known beforehand and therefore more generalized actors can be constructed, for example a Y operator which can perform the function of any of $Y_{\mu 1}$, $Y_{\mu 2}$, etc., depending on what is required, without forcing the user into the extra complication (and inefficiency) of our solution involving Y_μ. As a feature of a programming language, then, this extra flexibility would be of considerable merit. As a feature of a formalism for studying computation, such as the mu calculus, its value is more questionable. For the purposes of this research, then, we stick to the simpler definition of tokens given in axiom A3. Our approach to this problem resembles the use of Curried functions in the lambda calculus, whereby all computations can be expressed using only single-argument lambda-expressions.

Another issue regarding tokens is whether the full generality of being able to have arbitrarily long read and write tables is of any use. The driving force behind adopting that definition is that it simplifies the formal properties of tokens. If, for example, tokens were defined so that they could only be written once (as they are in Henderson[13]), but could be read any number of times, the question would arise

as to what to do if two attempts were made to write a given token. If the decision were made that the second write would "fail," then "second" would have to be defined, which would be difficult if the two events were not ordered with respect to each other by the causality relation. Any useful definition of "second" would introduce indeterminacy into the system, whereas the current token system is perfectly deterministic.

A second approach is to look at the uses we have found for tokens. The parallel evaluation examples generally involved a token being written at most once and read some number of times; the self-reference example involved reading once and writing many times (each time causing the read to complete again with a different value!); and the recursion examples given wrote once and read many times. No example was given where the read and write sides of a single token could each receive more than one message. However, a parallel evaluator (as in definition 2.8) for a multiple-valued applicative language such as Ward's EITHER-K theory[34] *would* use this property of tokens. This may be of limited interest in practical cases, but serves to add credence to the claim that the theoretical approach chosen includes the proper amount of generality.

The third angle from which tokens could be viewed is from the perspective of possible implementations. It will be seen in the next chapter that there are indeed reasonable ways of implementing the fully general token mechanism without excessive overhead.

2.5: Cells

While tokens extend the power of the mu calculus, they certainly do not enable us to model all interesting computations. In order to exhibit an actor implementation of a file system or a data base, some mechanism for causing side effects, that is, permanently changing the state of an object in a way that destroys information about previous states of the object, is almost essential. We shall argue at various points in this thesis that the explicit use of cells should be minimized, that other mechanisms with more confined kinds of mutability (such as tokens) can often be used more efficiently in distributed systems, but the fact remains that this approach will not suit all situations. Consequently, we introduce the concept of a *cell*[11] as the means by which arbitrary side effects may be achieved. Unfortunately, the introduction of cells adds considerable complication to the axiomatic description of the mu calculus, so we begin informally.

2.5.1: Informal Description of Cells

Somewhat in the same way as we viewed tokens, we view a cell as being composed of two *ports*: an *update* port u_x and a *contents* port c_x, where x is a unique identifier for that cell. The update port accepts messages with two components: a new value V and a continuation C. After updating the cell to have value V, the event (C) is caused. The purpose of including this continuation is that it can contain computations that should not begin until the update has been completed. The contents port of the cell simply accepts a continuation C and causes the event CV, where V is the current value of the cell. Similar to the τ operator for creating tokens, we postulate a ψ operator for creating cells, such that ψVC causes $Cc_x u_x$ for some previously unused cell identifier x. The initial value of the

cell will be *V*.

The notion of a cell having two ports as described above is somewhat unconventional; it is motivated by the desire that all transactions with a cell be by sending messages to it, rather than being mediated by special operators which we would have to invent. Even so, one can envision various devices by which an update message to a cell could be distinguished from a value message, but at the level of this work it is worth little to engineer such ad hoc solutions. A user who was annoyed by continually having to pass around both ports of a cell could easily design an actor which would package them into one unit. In any case, we make no claim that the structure presented here is superior, only that it is simple and adequate for our purposes.

2.5.2: A New Axiom Scheme for the Mu Calculus

In the foregoing sections, we have tacitly assumed that in a useful implementation of an interpreter for the mu calculus, every event that in theory could be caused by the initial set would eventually occur. In the absence of cells, such an interpreter would be deterministic in the sense that the same events would always result from a particular initial set of events. Cells, however, introduce the possibility of nondeterminism. If more than one ordering of the operations on a cell is possible, then different orderings may give rise to different results. This is not the same situation that arises in, say, handling of multiple actor bodies, which may result in, for example, R receiving the value 3 and also the value 4. This latter situation may arise as the natural consequence of some multiple-valued computation; the former refers to the possibility of an entire computation (including all concurrent events) taking one of two alternative routes *which are not consistent with one another*. For example, one might be characterized by the value of a particular

cell being 0, the other by the value of that cell being 1—it is not possible for a cell to have two values at the same time.

The introduction of objects such as cells in which state changes performed in different orders can produce different results requires some radical changes in our thinking. For example, it is no longer useful to talk about the closure E^* of an initial event set E as containing *all* the events that could be caused by events in E, because some of these events may be inconsistent with each other (e.g., be based on different assumptions about the value of a cell). Even if two event sets E_1 and E_2 each have closures which do not contain inconsistencies (i.e., lead to determinate computations), their union may not, due to interactions between cell operations performed by E_1 and E_2. Finally, in the presence of cells, it becomes possible to detect multiple executions of an event, leading to questions about the status of multiple copies of an identical event. Of course, all of this is just a rehash of the traditional problems encountered in trying to construct a sound semantic basis for a system incorporating both parallelism and side effects[11].

The approach we take here is based on a suggestion by Henry Baker that what is needed is not a relation on events, as is provided by our current axiom system, but a relation on partial execution histories, where an execution history would be said to cause all legal execution histories of which it was an initial substring. Instead of execution histories, however, we will consider system states, which are basically execution histories with the portions which are no longer relevant discarded. A system state will be expressed as a set of events; the axioms will describe what new states could be caused by the current state. Thus, for example, if a state S included the event

$$E \equiv (\mu xy.(Rx)(Ry))\ 3\ 4$$

then a new state S' that could be caused by S would be a state which contained all the events in S except for E, and additionally contained the two events R3 and R4 which are caused by E. The fact that E was part of the execution history is not retained in S' because it can have no bearing on any subsequent state transitions. We already have the consequents we want from E, and in a system with cells it would be dangerous to produce them again, so at best we could retain a copy of E in S' which was specially marked as already having been processed. Since we have no use for this information, we choose to delete it instead.

To avoid confusion with the event-causality relation -->, we denote the state-causality relation by ==>, using the related notation $==>^+$ for the transitive closure of ==> and $==>^*$ for the reflexive transitive closure. The new set of mu-calculus axioms (up to and including cells) is then

Axiom B1 *(mu-reduction)*:

If $E = ((\mu x_1 x_2 \cdots x_n . E_1 E_2 \cdots E_m)\ A_1 A_2 \cdots A_n) \in S$, then $S ==> (S-\{E\}) \cup S'$, where $S' = \{S[A_1, A_2, \cdots, A_n; x_1, x_2, \cdots, x_n; E_i] \mid 1 \le i \le m\}$.

Axiom B2 *(primitives)*:

If $E = (p c_1 c_2 \cdots c_n A) \in S$, where p is a primitive and each c_i is a constant, then $S ==> (S-\{E\}) \cup \{(A\ \tilde{p}[c_1; c_2; \cdots; c_n])\}$, where $\tilde{p}$ is the function denoted by p.

Axiom B3 *(tokens)*:

This axiom comes in two parts:

a) If $E = (w_x\ A) \in S$, then $S \Longrightarrow (S-\{E\}) \cup \{(w_{xy}\ A)\}$, where y is a new identifier that does not appear in any event in S.

b) If $C = \{(r_x\ A)_{a_1 a_2 \cdots a_m}, (w_{xy}\ B)\} \subseteq S$, then $S \Longrightarrow (S-C) \cup \{(r_x\ A)_{a_1 a_2 \cdots a_m y}, (w_{xy}\ B), (AB)\}$, provided that y does not appear on the list $a_1, a_2, \cdots, a_m$.

Axiom B4 *(creation of tokens)*:

If $E = (\tau C) \in S$, then $S \Longrightarrow (S-\{E\}) \cup \{(Cr_x w_x)\}$, where x is a new identifier that does not appear in any event in S.

Axiom B5 *(updating cells)*:

If $C = \{(u_x\ A)', (u_x\ BC)\} \subseteq S$, then $S \Longrightarrow (S-C) \cup \{(u_x\ B)', (C)\}$.

Axiom B6 *(reading cells)*:

If $\{(u_x\ A)', (c_x\ C)\} \subseteq S$, then $S \Longrightarrow (S-\{(c_x\ C)\}) \cup \{(CA)\}$.

Axiom B7 *(creation of cells)*:

If $E = (\psi VC) \in S$, then $S \Longrightarrow (S-\{E\}) \cup \{(u_x\ V)', (Cc_x u_x)\}$, where x is a new identifier that does not appear in any event in S.

2.5.3: Discussion

Axioms B1 and B2 are fairly straightforward adaptations of axioms A1 and A2 to use the state-causality concept already discussed. Axiom B3, however, contains enough complication to bring out most of the subtle points of the new scheme. The first point concerns duplication of events. Strictly speaking, the states referred to above are not sets of events, but rather *collections* of events (in the sense defined by Hewitt for PLASMA[15]) in which duplicates are allowed. If some state contained two copies of the event R3, we assume that the intent was in fact that the value 3 should be printed twice. For a more sophisticated example, consider a state containing two copies of the event

$$c_x(\mu a.\sigma a(\mu b.u_x b(\mu.Rb)))$$

In this case, it is at least plausible that the cell *x* might be incremented twice (although without any kind of mutual exclusion operator we cannot be sure this will be the result). In any case, a state with two copies of that event is clearly a different entity from a state with some other number of copies. Thus the state union S∪T denotes a state containing a copy corresponding to every event in S as well as a copy corresponding to every element in T, and the state difference S-T denotes a state which is like S except one copy of every event in T has been removed.

Given our concern that the correct number of copies of each event should appear in a state, tokens clearly present a problem. An incorrect possibility for axiom B3 would be to adapt axiom A3 to the state model, yielding, "If $\{(r_x\ A),(w_x\ B)\} \in S$, then $S \Longrightarrow S \cup \{(AB)\}$." Unfortunately, this could lead to an unbounded number of copies of the event (*AB*) being added to S, where we intended that only one should be added. The situation is further complicated by

the fact that if S contained, say, one copy of $(r_x\ A)$ but two copies of $(w_x\ B)$, then we would want exactly two copies of (AB) to be generated. Our desire to ensure that exactly one event (AB) will be generated for each pair $\{(r_x\ A),(w_x\ B)\}$ that exists in S leads us to tag one of the events in the pair (we have picked $(w_x\ B)$, though the choice is completely symmetrical) with an additional unique identifier y, so if in fact there are two copies of $(w_x\ B)$ in S, each will eventually be assigned a different additional identifier before any other use can be made of the event. Thus part (a) of axiom B3 removes this possible source of ambiguity by eliminating the possibility that two distinct events in which w_x is sent some object could look identical.

Part (b) of axiom B3 contains the mechanism by which the token actually does its work. The events that come into play here are the write-event $(w_{xy}\ B)$ generated by part (a) from $(w_x\ B)$ and the read-event $(r_x\ A)_{a_1a_2\cdots a_n}$. The (initially empty) list of subscripts $a_1a_2\cdots a_n$ is the list of additional identifiers y of the write-events with which the read-event has already interacted. This list is kept to ensure that no read-event ever interacts with the same write-event more than once. Since the identifiers appearing in this list are unique among all write-events, the proper semantics will be maintained for multiple copies of the same object sent to the same token. In particular, if some state contained two copies of the event $(w_x\ B)$, some subsequent state would instead contain the two *distinct* events $(w_{xy}\ B)$ and $(w_{xz}\ B)$, and each would thus have one chance to interact with each read-event $(r_x\ A)$. Consequently, two copies of the event (AB) would be generated from each such read-event, as desired. When a read-event $(r_x\ A)_{a_1a_2\cdots a_n}$ interacts with a write-event $(w_{xy}\ B)$, not only is the consequent (AB) generated, but the read-event is replaced in the subsequent state by $(r_x\ A)_{a_1a_2\cdots a_ny}$—that is, a copy of the read-event with y appended to its list of subscripts, preventing that

read-event from ever interacting with the same write-event again.

Axiom B4 is much more straightforward—it simply provides for the replacement of the token creation event (τC) by its intended consequent ($Cr_x w_x$), where x is the unique identifier to be assigned to the token.

Axioms B5, B6, and B7 deal with cells, and use a special convention to record the current contents of a cell in the state. The presence of the primed "event" $(u_x\ A)'$ in a state S indicates that the value of cell x in state S is A. The axioms are written so that for every cell known in S there is at all times exactly one such value-event. In light of this convention, the cell axioms are seen to be quite simple. An update event $(u_x\ BC)$ simply replaces the old value-event for the cell with the new value-event $(u_x\ B)'$ and generates the appropriate consequent (C) as an indication that the update has been performed. A contents event $(c_x\ C)$ causes the current value of cell x to be sent to C. Finally, the cell-creation event (ψVC) causes the initial value-event $(u_x\ V)'$ along with the consequent $(Cc_x u_x)$ containing the names of the two ports of the cell.

2.5.4: Congruence of States

Just as the notion of congruence (as defined in definition 2.3) was useful for determining whether or not two superficially different objects or events were in fact functionally distinct, it will be convenient to define a congruence relation on states. Obviously it is convenient to call two states S and T congruent if they are collections of congruent events: in other words, if there is a one-to-one correspondence between events in S and events in T which matches every event in S to a congruent event in T (where congruence is as defined in definition 2.3).

There is another kind of congruence between states which is more subtle, however. This kind of congruence has to do with the unique identifiers used for tokens

and cells in this scheme. Ths only item of significance about these identifiers is that every instance of the same cell (or token) bears the same identifier. The text of the identifier carries no information in itself; it only acquires meaning from the relationships between the various contexts in which it appears. Thus these identifiers are not all that different from bound variables of actors, except that there is no one place (corresponding to the bound variable list of an actor) where they are explicitly "bound." This observation motivates us to consider another kind of congruence, one in which two states S and T differ only in the choice of unique identifiers for cells and tokens. Thus we state

Definition 2.9:

Two states S and T are *congruent* if there exist functions *f* and *g* with the following properties:

1) *f* is a one-to-one function from unique identifiers appearing in S to unique identifiers appearing in T.
2) *g* is a one-to-one correspondence between events in S and events in T with the property that each event in S corresponds to an event in T which is congruent (in the sense of definition 2.3) except that any unique cell or token identifiers occurring in the former event have been replaced in the latter event by their images under *f*.

2.5.5: Conclusions Regarding the State Model

This section has presented a new approach to the semantics of the mu calculus, based on the concept of the global state of the system rather than strictly on the consequences of individual events. This change in viewpoint was stimulated by the difficulty of handling cells using the old model. This, in turn, seems to be because *cells in fact contain global information*, which can be used as a medium for communication between two computations, however distant from each other, which share the same cell. The necessity of taking this view already portends some trouble in implementing a fundamentally centralized concept such as a cell on a distributed system. This will be dealt with in more detail in the next chapter. A subsequent section of this chapter, however, will be devoted to arguing that there are better ways of writing many programs which we currently write using side effects, especially if these programs are to be well suited for a distributed system. Further discussion of these considerations is postponed until that section.

Finally, the state model presented here violates several desiderata that have been laid down for semantic bases for concurrent computation, as by Hewitt and Baker[16]. Sound arguments are made there that a theory of computation should be based on a local, rather than a global, view. In fact, Hewitt uses the analogy to relativity theory to argue that a global state is a figment, an unobservable and useless concept. Along these lines, an interesting property of our state model should be noted. If a state S can be *partitioned* into two states S_1 and S_2, and if $S_1 ==>^* T_1$ and $S_2 ==>^* T_2$, then $S ==>^* T$, where $T = T_1 \cup T_2$ (assuming no conflict between unique identifier generators used in processing S_1 and S_2). The converse is of course not true, but this property shows a way of decomposing a global description of a computation into successively more local views. In fact, this is the theoretical basis for our subdividing the state S among various processors and

running the computations more or less independently. Of course, an event in S_2 may occasionally need to interact with an event in S_1 in order to produce some result, but this is just another way of saying that the processors will have to communicate. Every now and then, events may logically need to be transferred from one processor's responsibility to another's in order for the system to continue operating.

Thus although the global state S may indeed be a figment, the local states S_1 and S_2 may not. What we then know is that the hypothetical state S always behaves as if it were composed of some interleaving of the changes to S_1 and S_2. Many concerns in program specification and verification, though, can be addressed by considering only a small substate of S directly related to the problem at hand. If this semantic model is pursued, more work may have to be done on the significance of transferring events from one of these local spheres to another, but in any case the model does seem suited to studying the local behavior of small subcomputations as well as examining the global behavior of entire systems.

2.6: General Synchronization Operators

A weakness that may be felt in some applications remains in the mu calculus even after adding all the features described in the previous sections. This weakness is the inability to specify in a general way any kind of synchronization operation involving mutual exclusion. It is not the purpose of this section to deal in any deep way with the voluminous literature that has been devoted to synchronization problems, nor to discuss all the solutions that have been proposed. Rather, an attempt will be made to show, by example, that solutions to such problems can fit naturally into the state semantics described in the previous section. Thus it is entirely reasonable to imagine writing, for example, a mu-expression that will serve

as an arbiter, imposing an ordering on otherwise uncoordinated requests to access a resource. Of the many *synchronization primitives* that have been suggested by various authors, we pick Dijkstra's *semaphores*[7] as being a serviceable and familiar set of primitives for our demonstration.

2.6.1: Semaphores

We can imagine a semaphore as a pair (p_x, v_x) of related objects, just as tokens and cells were treated. An event of the form $(p_x C)$ performs a P operation (a request to proceed) on the semaphore and, when the conditions are right, causes the event (C). An event of the form (v_x) will result in a V operation being performed on the semaphore. To create new semaphores, we can use the special actor Σ; an event such as (ΣC) will result in the creation of a new semaphore (p_x, v_x) and cause the event $(C p_x v_x)$. The new semaphore will be initialized so that the number of P operations completed can never exceed by more than one the number of V operations completed. More formally, we can state the following state-model axioms for semaphores.

Axiom B8 *(P operation)*:

If $C = \{(v_x),(p_x C)\} \subseteq S$, then $S \Longrightarrow (S-C) \cup \{(C)\}$.

Axiom B9 *(creation of semaphores)*:

If $E = (\Sigma C) \in S$, then $S \Longrightarrow (S-\{E\}) \cup \{(C p_x v_x),(v_x)\}$, where x is a new identifier that does not appear in any event in S.

Note that no explicit axiom is needed for the V operation.

The operation of these two new axioms is fairly simple. The presence of an event (v_x) in a state S signifies that a P operation in that state can succeed. Thus the value of the semaphore (p_x,v_x) in a state S is simply the number of events (v_x) in S minus the number of events in S of the form ($p_x C$). When a P operation occurs, it annihilates an event (v_x), preventing other P operations from completing (unless other events (v_x) remain). When mutual exclusion is no longer necessary, a V operation will restore the event (v_x) so that another P operation may proceed.

In order to meet the specifications described informally earlier, the creation of a semaphore must entail the creation of an initial event (v_x) so that one P operation can complete before any V operation is begun. Axiom B9 takes care of this requirement.

2.6.2: Construction of an Arbiter

As an example of how semaphores may be used in the mu calculus, we consider the construction of an arbiter actor α. The behavior we desire is that an event of the form (αFC) cause an event (CF^*), where F^* is a version of the actor F enclosed in an arbiter; in other words, an event (F^*XY) will cause an event (FXY^*)—where Y^* is a continuation derived from Y—but *only* if no other such event has been caused and not yet resulted in F returning an answer to its continuation. Thinking of F as a function, we see that access to F will be *serialized*: no message to F^* will be passed along to F if any invocation of F is currently active. This kind of behavior would be desirable if F were, for example, a function that allocated seats on a particular flight in an airline reservations system.

An arbiter with the desired properties can be defined as

$$\alpha \equiv \mu fc.\Sigma\mu pv.c\mu xy.p\mu.fx(\mu z.(yz)(v))$$

As an example, let us imagine applying this actor to some actor *F*. Our initial state is then

$$S_0 = \{(\alpha FC)\}$$

This will cause these subsequent states:

$$S_1 = \{(\Sigma\mu pv.C\mu xy.p\mu.Fx(\mu z.(yz)(v)))\}$$

$$S_2 = \{((\mu pv.C\mu xy.p\mu.Fx(\mu z.(yz)(v)))p_i v_i),(v_i)\}$$

$$S_3 = \{(C\mu xy.p_i\mu.Fx(\mu z.(yz)(v_i))),(v_i)\}$$

$$= \{(CF^{\pi}),(v_i)\}$$

Let us suppose that due to machinations inside *C*, S_3 eventually causes

$$S_4 = \{(F^{\pi}X_1Y_1),(F^{\pi}X_2Y_2),(v_i)\}$$

Among the states that may be caused by this is

$$S_5 = \{(p_i\mu.FX_1(\mu z.(Y_1z)(v_i))),(p_i\mu.FX_2(\mu z.(Y_2z)(v_i))),(v_i)\}$$

which, by axiom B8, can lead to either of

$$S_6 = \{(\mu.FX_1(\mu z.(Y_1z)(v_i))),(p_i\mu.FX_2(\mu z.(Y_2z)(v_i)))\}$$

$$S_6' = \{(p_i\mu.FX_1(\mu z.(Y_1z)(v_i))),(\mu.FX_2(\mu z.(Y_2z)(v_i)))\}$$

Note that, in either of these, the event that still has p_i as the receiver will remain untouched in all future states until an event (v_i) is added back. Since the situation is symmetrical, let us pursue the consequences of S_6.

$$S_7 = \{(FX_1(\mu z.(Y_1z)(v_i))),(p_i\mu.FX_2(\mu z.(Y_2z)(v_i)))\}$$

$$S_8 = \{((\mu z.(Y_1 z)(v_i))Z_1),(p_i \mu.F X_2(\mu z.(Y_2 z)(v_i)))\}$$

assuming F returns the result Z_1 to its continuation when sent the value X_1.

$$S_9 = \{(Y_1 Z_1),(v_i),(p_i \mu.F X_2(\mu z.(Y_2 z)(v_i)))\}$$

Now that the invocation of F resulting from $(F^{\pi} X_1 Y_1)$ is complete, the event (v_i) is added again so that the pending request can proceed in a similar fashion. At no time, however, was it possible for both to be active at once.

It is significant that in the use of the arbiter α, the argument F must be not just an actor but a *function*—an actor which indicates that it is "done" by following the convention of returning a single value to its continuation (similar requirements attend the use of the parallelism actor π discussed earlier). If the actor F *never* returns an answer to its continuation, then the semaphore will never be released, and no other request to F will ever be allowed to proceed. If F returns more than one answer to its continuation, as in the "function"

$$F \equiv \mu xc.(c3)(c4)$$

then two V operations will be performed on the semaphore, which will thereafter operate incorrectly (allowing *two* or more invocations of F to be active at the same time). The requirement that the object subject to arbitration obey the ground rules for a function is not unique to this scheme; it applies also, for example, to Hewitt's *serializer* construct[14].

2.6.3: Conclusions Regarding Semaphores

After a brief discussion of the problem of mutual exclusion, semaphores were introduced as a way of implementing mutual exclusion in the mu calculus. The goal was not to shed any new light on synchronization problems, but simply to demonstrate the possibility of including mechanisms for dealing in a natural way with mutual exclusion in the mu calculus. A simple application of semaphores, the construction of an arbiter to serialize access to an actor, was then explained, to show how semaphores might be used in the mu calculus. Finally, some aspects of the arbiter, including restrictions on the class of actors to which it can be applied, were discussed.

Not treated in this brief overview were several of the thorny synchronization problems described in the literature. Various problems have been proposed which cannot easily be solved using semaphores; however, there is no reason to believe that it would be difficult to include other synchronization primitives in the mu calculus. Another unexplored dimension concerns fairness of scheduling in synchronization operators. There are solutions, for example, to the readers-writers problem[11] which are partially correct—no violation of the desired mutual exclusion can occur. Nevertheless, it is possible for an unfortunately-timed sequence of, say, read requests, to prevent a write request from ever completing, if the semaphores used are "unfair." A fair semaphore, on the other hand, would honor requests in the order of their arrival; thus any request is assured of being processed after a finite length of time.

The natural question to ask is, are the semaphores described by axioms B8 and B9 fair or unfair semaphores? One answer is that it is impossible to tell. The state model gives, for some initial state, a set of later states that it could cause. It does not say which of these possibilities would be produced by an actual

implementation. The formulation given for semaphores does not exclude the possibility of the semaphores being fair—all possible outcomes from a fair semaphore are represented. However, if we interpret our state model as representing only those implementations which could conceivably generate *any* of the consequences predicted by the state model, then we must answer that the semaphores described are unfair, since a given initial state may cause some states which could only be reached in an implementation that had unfair semaphores.

Interestingly enough, there is no simple way to fix this in our model. One could modify the semaphore axioms so that the "oldest" P request in a state is always the next one processed, but this would be only a partial solution. That is because there is no guarantee of fair scheduling in our model, no guarantee that a state cannot cause an infinite sequence of other states, without some event in the original state ever being processed. Once again, our model does not exclude fair scheduling, but it does not specify it, and without a guarantee of fair scheduling a fair semaphore is meaningless. The conclusion to be drawn from all of this, then, is that our state model is useful for describing all possible results of a computation, but less useful for distinguishing the results that some particular strategy, such as fair scheduling, would allow, from the results possible under some scheme offering fewer guarantees.

2.7: Coding Imperative Constructs

So far, our attention has been focused primarily on developing various mechanisms for the mu calculus and showing their relationship to languages with a mainly applicative flavor, such as LISP and the lambda calculus. Grudging acknowledgment has been made in the previous two sections of the existence of side effects and the desirability of being able to model them, but no discussion of imperative styles

of programming has been conducted. This has been due, first, to the fact that applicative constructs are usually easier to deal with formally, and second, to a prejudice that side effects are difficult to handle in distributed systems and therefore are best avoided. The prejudice remains; this author is convinced that extensive use of side effects (such as cells and semaphores) will make it much more difficult for a distributed system of the kind envisioned to process a program at top efficiency. Since so much of the world programs in a basically imperative style, however (using languages such as FORTRAN, COBOL, PL/1, ALGOL, PASCAL, etc.), the goal of this section is to argue that many activities of imperative programs (such as assignment statements) that are ordinarily thought of as being accompanied by side effects, can be expressed in a form that is quite free of side effects and thus much more suitable for a distributed system.

The mechanics of translating programs written in an imperative style (including assignments, go-tos, conditionals, looping, and parallelism) into a continuation- or message-passing style have been explored in great depth by Sussman and Steele[29,30,33]. Their translation includes the removal of side effects caused by most assignment statements and is quite applicable to producing mu-calculus expressions as output. It would be repetitive and inappropriate to go into such detail here, but some aspects of our view of parallelism specifically and message passing generally warrant some further discussion.

The Sussman-Steele scheme works by translating a sequence of statements into a continuation-style program, where the first statement in a sequence becomes an event containing the translation of the remaining statements inside a continuation actor which is part of that first event. Many side effects, such as those associated with most assignments, can then be eliminated by viewing local variables not as cells into which different values may be put over time, but as *parameters* to the

continuation. Thus the translation of statement *n* receives as parameters the values of all local variables, plus a continuation actor containing the translations of all statements following *n*. If the execution of statement *n* has no effect on the value of any local variable, it will eventually pass to the continuation exactly the set of local variable values it received. If statement *n* is, for example, an assignment to local variable *x*, then no side effect is required. Instead, the set of local variable values passed to the continuation will have the new value for *x* in place of the old one. Sussman and Steele accept that not all assignments can be, or should be, treated in this fashion—some assignments are to "global" variables: for example, directories in a file system. This is where the concept of cells should fit in. It is nevertheless true that the Sussman-Steele procedure can be used to translate ordinary imperative programs into a form which is largely free of side effects.

Assuming that sensitivity to side effects has been removed from some section of a program, it can then be adapted in various ways to be even more suitable for execution on our kind of distributed system, especially with appropriate choices in the design of the original imperative language. To echo many other researchers[2,12], it should be possible to engage in parallel evaluation of subexpressions, for example, of *A* and *B* in the expression "*A*+*B*," or even in the case where *A* and *B* are procedure arguments, as in "f(*A*,*B*)." Our token mechanism seems a quite serviceable way of accomplishing this. Another way of increasing parallelism is through multiple assignment statements such as "a,b,c := *A*,*B*,*C*" where *A*, *B*, and *C* should all be able to be evaluated in parallel. This fits in very naturally with the imperative- to continuation-style translation described above.

A third possible way of increasing the power of an imperative language to express parallelism might be by a construct such as "*S*&*T*," where *S* and *T* are statements. The semantics of this statement would be that *S* and *T* may be

executed in parallel, and that the following statement may not begin execution until both *S* and *T* have finished. The "S&T" statement would allow the programmer great freedom to express a wide variety of permissible orderings of computations; this flexibility could then be used to maximum advantage by the system. It is not hard to see how the semantics of this statement could be constructed using tokens, although problems do arise if both *S* and *T* modify local variables and those changes must be merged at the end. More serious problems arise if *S* and *T* use assignments to local variables for communication; in this instance cells would probably have to be used to implement those variables.

Even more exciting possibilities for generating large amounts of parallelism are present in the looping constructs of imperative languages. In many cases such loops are used simply to express the notion that some operation is to be performed on every element of some data structure. If the order is unimportant, which is frequently the case, a looping construct allowing all activations of the loop body to proceed in parallel, rather than sequentially, could result in large amounts of useful parallelism. Imagine, for example, a compiler simultaneously performing type checking or even translation of every statement in a block. An organization that allowed all those activities to proceed in parallel might well be able to make use of several processors to speed the compilation.

The discussion in this section is necessarily of a very speculative nature, since the rigorous detail beneath it, if developed carefully, would expand in size out of proportion to its place in this thesis. Nevertheless, it is hoped that some reasons have been presented to be optimistic that the mechanisms for parallelism that have been developed in this chapter can be used to generate large numbers of parallel activities performing programming tasks of general interest. In the next chapter, we will be banking on this property as the key to being able to apply

multiple processors to such tasks. The theory will be that, even if at any time the *majority* of parallel activities are blocked, awaiting some information from a distant site, there will still be several activities at any time that are not blocked, and thus several processors can be kept busy.

2.8: Summary

This chapter has outlined the mu calculus, a formalism for studying message-passing computation. The development began with the pure mu calculus, which is very much like a restriction of the lambda calculus, and proceeded through the addition of tokens, cells, and semaphores. Along the way, correspondences between the applicative, imperative, and message-passing styles of programming were discussed. Also explored were applications of the mu calculus to implementing recursion, list structures, self-referential structures, arbiters, and other constructs.

In spite of these "applications," the mu calculus is not billed as a programming language. Others[15,19] have developed programming languages based on message passing. In contrast, this chapter has sought to develop a formalism, bereft of all embellishments, for better understanding the essence of message passing. For this reason, the mu calculus has been made as spare a language as possible. If it is to be used directly at all, one can only imagine it as the "machine language" for a distributed system, perhaps the one outlined in the remainder of this thesis. Actual programming would certainly be done in a highly sugared version of the mu calculus or in a different kind of language altogether, which would then be translated to a form resembling the mu calculus. The various translation rules presented suggest some ways in which this desugaring or translation might be

done.

As an example of "the essence of message passing," the mu calculus invites controversy. Several aspects of message passing that are manifest in, say, Hewitt's PLASMA[15], are absent from the mu calculus, even disregarding the extensive syntactic sugaring that goes into making PLASMA a usable language. For example, PLASMA incorporates the notion of a process, whereas the mu calculus does not. The advantages of recognizing the existence of processes include the ability to treat a process as a object, and consequently to perform operations on it. In a practical system, this provides a framework for keeping various useful pieces of information, such as a process's user ID, priority, or whatever. It also permits a convenient mechanism for identifying and killing runaway computations. On the other hand, the concept of processes is one that, as we have seen in this chapter, we can do without, simplifying our formalism. Whether the simplification removes something essential from the formalism, as far as its usefulness in modeling real situations is concerned, remains an open question.

Also left out of the mu calculus is another part of the actor ideology. As articulated by Hewitt, this ideology holds that all transactions, including message transmissions, are mediated by actors. Thus the causation of an event entails the arrival of a messenger actor in the vicinity of a target actor. This messenger actor is a package containing the objects actually being sent to the target. The target may then extract these objects by sending messages to the messenger, which may in turn send messages to these messages, and so on. (It should be pointed out that Hewitt makes good use of the packaging of messages into messengers to implement forms of keyword-based, as well as position-based, conventions for supplying arguments to actors.) It is recognized that at some point this potentially infinite regress must be terminated, so it is stipulated that at some point a

message may be examined by accessing its innards directly, rather than by sending it messages and awaiting its replies. The mu calculus eliminates this regress by explicitly providing a basis for it. In the mu calculus, actor arguments are not packaged into messenger actors, but instead are immediately available. The user may, of course, build on top of this as many layers of message-transmission protocol as desired.

This discussion of messengers is related to another important part of the actor ideology. This is the dictum that, as far as is practicable, objects in an actor system should interact with each other by exchanging messages, rather than directly accessing each others' innards. Thus, for example, an actor which computes the negative of its argument should send a message to its argument asking it what its negative is, rather than trying to examine the argument and compute the negative that way. Adherence to this dictum can increase the independence of operators from the representations used for data, and also facilitate the extension of operators to new data types. Although the arithmetic operators of the mu calculus have not been specified to work this way, this has been because it was desired to maintain some similarity to the lambda calculus and, once again, to provide a basis—somewhere there has to be some object which knows how to perform basic arithmetic operations directly. The design of the mu-calculus primitives was certainly not an attempt to take a position on this aspect of the actors approach, which has much to recommend it. As before, the user is free to add levels of protocol on top of the bare mu calculus to achieve this end. Exploration of these issues is not a major goal of this thesis, however.

The mu calculus is thus designed to serve two purposes. One is to gain a better understanding of the essence of the message-passing style of computation by excising all that seems inessential. The other is to serve as a semantic basis

for programs running on a distributed system. We now turn to the task of developing a multiprocessor implementation of an interpreter for the mu calculus.

Chapter 3: Implementations

The goal of this research has been not only to specify a language (the mu calculus) with the potential to make good use of a distributed system, but to demonstrate that potential by describing an appropriate implementation. This chapter will therefore be devoted to outlining an implementation of the mu calculus.

3.1: The Basic Approach

3.1.1: Semantic Structure

The implementation we choose will be based on the concept that there exist a variety of objects scattered around a multiple-processor system. These objects may move from one site to another as the needs of the system require, and, under suitable circumstances, multiple copies of an object may be maintained. Objects are not monolithic, but contain structure, depending on the type of data the object is supposed to represent. Among the kinds of components an object may have are references to other objects; thus the system will have the capability to support tree-like or even cyclical organizations of data. The semantic properties of object-reference systems such as this have been studied at some length[4] along with possible implementations on centralized computer systems.

In our case, objects might be actors, cells, tokens, numbers, or other kinds, perhaps specialized to a particular application (a graphics system, for example, might include three-dimensional vectors and 3x3 matrices as primitive object types, along with the applicable primitive operators). Since many of these kinds of objects (e.g., actors) may have a great deal of structure, the ability of objects to naturally reference other objects is crucial.

In addition to objects, our system must have some way of representing events.

Events, however, are never more than a transitory presence in our system—they serve only to mark the current place in a computation until that computation advances one step, leading to the creation of new events and the demise of the old one. Thus it is profitable to treat events somewhat specially.

It is useful to think of a system-wide *event list* containing all events in the system that have been created but not yet processed. This global event list corresponds closely to the global system state described in the previous chapter. In practice, this system-wide event list will be distributed (partitioned) into several smaller event lists, one maintained by each processor in the system.

In a more traditional system, the function served by the event list might be served instead by a table of active processes. Typically, each process in such a table would contain various pieces of state information about the process, such as saved register contents, stack pointer, process stack, location of the core image for the process, etc. In the message-passing system described here, all this information is contained in the selection of objects referenced by an event. There are no processes per se—just events waiting to be acted upon.

3.1.2: Physical Structure

Any student of distributed systems quickly comes face to face with a whole variety of different possible network structures: rings, Ethernets, shared buses, hierarchical organizations, store-and-forward networks, etc. Many of the properties of these are similar, but there are differences in capabilities relating to the ease of knowing whether a message has been received, bandwidth, response to contention for access to the network, routing of messages, and other matters. As much as possible, it is the intent of this research to avoid becoming committed to the narrow technological characteristics of any one type of network. However, for

concreteness, it will be helpful to make certain assumptions. Furthermore, the algorithms presented depend on a certain *logical* organization of processors which may be more easily achievable with some physical organizations than with others. This section seeks to outline a physical organization which is very compatible with the required logical organization, and make some preliminary comments on which aspects of its design are essential for the proper functioning of the algorithms to be presented below. Additional comments on this topic will appear throughout this chapter as the relevant concepts are discussed.

The basic logical structure assumed by our implementation will be a collection of processors, each with a private local memory (i.e., no sharing of memory between processors is required), each connected to a small number of other processors which are its *neighbors*. The connections are bidirectional and symmetrical; thus if A is a neighbor of B, then B is a neighbor of A. Such an organization of processors is thus equivalent to an undirected graph (which may or may not contain cycles) in which only a few arcs emanate from each node. "Few" here is a relative term; its use is due to the fact that the overhead incurred by a processor will increase if that processor accumulates more neighbors. Thus it might well be appropriate for higher-capacity machines or machines with more of a commitment to serve the network (rather than, say, their owners) to have a greater number of neighbors. A variety of topologies are consistent with these general restrictions (some are shown in Figure 3.1).

In addition to the specifications given above, each processor will be required to have a processor ID unique to the entire system (or some other way of generating names guaranteed to be unique throughout the system). It is not necessary, however, that all processors be identical, or that every processor use the same external data representation in communicating with its neighbors. In general, a

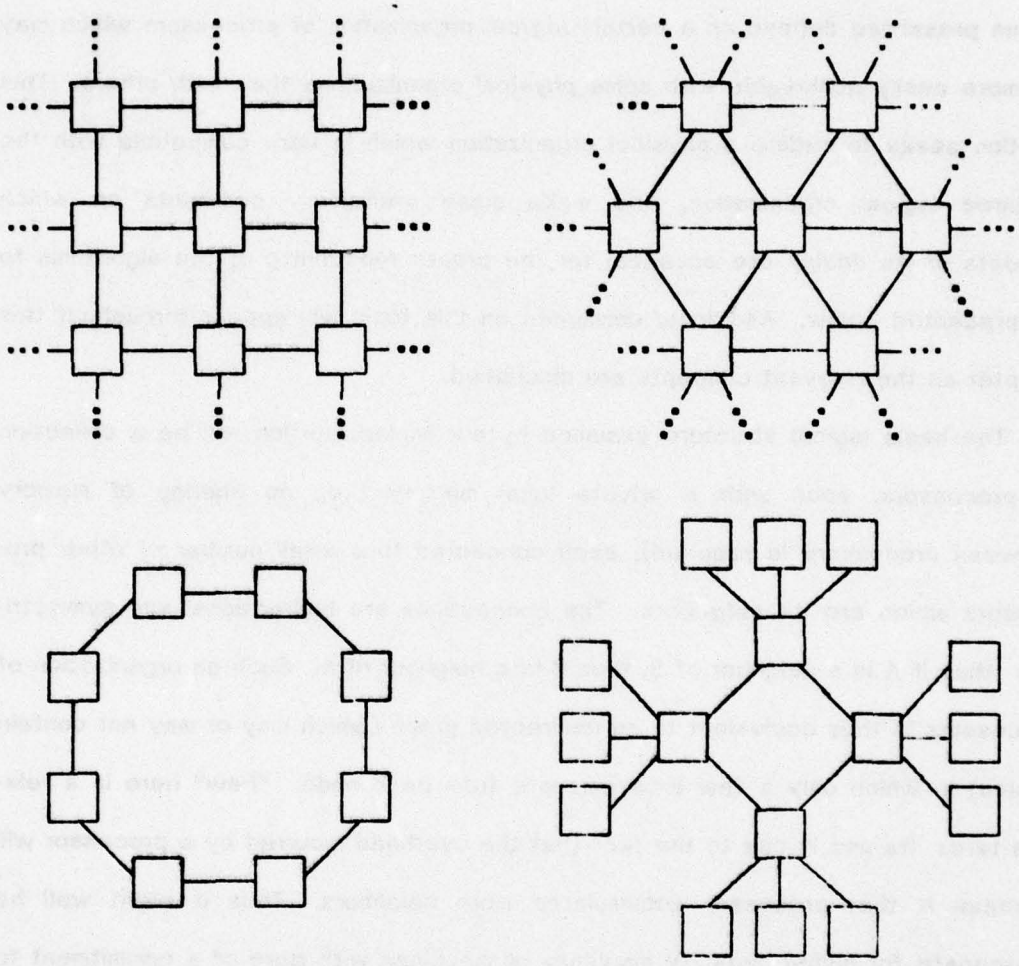


Figure 3.1: Some possible network topologies

different representation could be used for each link, subject to a few provisos. First, obviously the processors at each end of the link must understand the representation used on that link. Second, the representation must be sufficiently general that any kind of object that may exist in the system can be passed over the link. Third, the link must support transmission of the system-wide unique names of objects in such a fashion that the name of an object will always be recognized, even if it arrives at a processor via two different routes. Fourth, although in theory

the protocols used on different links may differ, they are all likely to have the common features described in this chapter.

The physical topology of the system need not follow the logical topology described above. The logical topology in effect constrains the paths over which information may travel; any physical topology which permits information flow over these channels may form the basis for an acceptable implementation. For example, an Ethernet (on which every processor can communicate directly with every other processor) could be made to support the logical topology described above either by declaring every processor to be a neighbor of every other (which might however impose a large amount of overhead on each), or by choosing for each processor a set of logical neighbors, and constraining the communication patterns on the net so that no processor ever sends a message to another processor that is not its neighbor. This approach makes less than full use of the capabilities of the Ethernet, since a processor might occasionally be forced to follow a rather tortuous path to communicate with another on the same net, but might reduce the overhead within the processors for object management, as we shall see.

In any case, the scheme presented here was certainly not designed for Ethernets, but rather for physical networks with properties closely matched to the logical network. Such networks have several advantages:

* expansibility. The network can be expanded to a very large size at little marginal cost. The space allocated for unique processor ID's does grow, but only logarithmically. Otherwise, expansion presumably involves simply hooking up new processors to the edges of the network, and has only a very local impact.

* bandwidth. For many topologies, there are potentially a large number of

communication paths between any two points in the network; no central Ether or other medium serves as a bottleneck.

- reliability. Even a catastrophic hardware failure at some node is likely to affect only a limited number of other nodes (its neighbors). In systems with a central medium, there are components whose failure will stop all communication on the network. Of course, reliability is also strongly influenced by the software system's ability to carry on in the face of failures; admittedly, this thesis does not address this problem very thoroughly.

- flexibility. Many different kinds of processor and link technology can in principle coexist in the system, allowing considerable freedom in picking the lowest-cost option for the performance desired at each point.

- other advantages that will become evident when the object-management algorithms are discussed.

There are of course disadvantages to this kind of organization also. Chief among these is the need for extra processors to become involved in transactions between processors which are not neighbors, with the attendant overhead and delay. It is hoped that our scheme will tend to minimize the need for this kind of transaction, but it remains to be seen whether the advantages cited above are worth this drawback.

3.2: Overview of System Operation

3.2.1: Objects and Object References

As was mentioned in the previous section, we desire to allow objects to refer to other objects; by using such a reference, an object may logically "contain" another object without physically containing it. These references will also allow sharing of objects, as well as increasing the structural modularity of the data, as has already been discussed. In order to implement this, we introduce the concept of an *object reference*[4], which is an entity that uniquely identifies a particular object without necessarily containing explicitly all the information in the object. In machine-language programming, a pointer can be such a reference: it serves to uniquely identify the area in storage containing the desired information, yet inspection of the bits of the pointer will not yield that information. Although it is not necessary, we may imagine that all object references have the same length; thus a reference to any object will in principle fit into the place of a reference to any other object.

The first tradeoff encountered in designing a system using object references involves deciding what to include in the reference. At one extreme, an object reference might be just a unique identifier for an object (like a pointer to a memory address). At the other extreme, large amounts of information about the object might be included (type code, length, hash code, etc.—naturally, such information can only be kept in a reference if it remains valid through all operations that may be performed on the object). The purpose of including this additional information would be to enable some operations involving the object (such as determining its type) to be performed using only the reference, without needing access to the full text of the object. However, the inclusion of extra information increases the amount of space required for an object reference. In this space-time tradeoff, the

space required for the extra information in the reference will be more worthwhile if the amount of time required to access the object from a reference is longer. In a distributed system, where the object may even need to be fetched from another site, it seems well worthwhile to include extra information in object references.

In any case, we see that an object in general will have two components: a reference (which if the object is shared may exist in several places) and a *text* which contains all information about the object not contained in the reference (simple objects, such as numbers, may not have any text, if all information associated with the object can fit into the reference). There is conceptually only one copy of the text, shared by all possessors of a reference to an object. Obviously, if side effects can be performed on an object (such an object can be termed *mutable*[20]), all such effects must be accomplished by modifying the text, which is the shared portion of the object. If an object cannot be changed (an *immutable* object), then many copies of its text may be made, so that a copy may be kept near each site where it might be needed. Even a mutable object may have multiple copies of its text made under appropriate circumstances, a topic to which we shall return.

For the sake of concreteness, we may imagine an object reference to contain the following fields:

- a unique identifier. This field is determined when the object is created and is different for each distinct object in the entire system. It might, for example, be constructed by concatenating the unique processor ID of the processor where the object was created and the count of the number of objects created on that processor up to that time. It is important to note, for reasons that will be explained later, that this implies no special responsibility of

this processor for the object once it has been created.

- a type code. This field distinguishes between actors, strings, numbers, and other primitive types in the system.
- a length. This is primarily useful for planning message transfers involving the text of the object, but could also be used as a first cut in, say, checking the equality of two strings.

An object reference containing the above information could fit comfortably into 64 bits, assuming a modest-size system. This allows for about forty bits of unique identifier. A larger system might need more space for unique identifiers, but it is hard to imagine that, say, 100 bits would not suffice for this purpose.

The object reference format just described is one that would be suitable for communication between machines. Inside each processor, various tricks obviously can be used to reduce the amount of storage space required (and also facilitate access to other information about the object). It is convenient to convert incoming object references at a site by looking them up and/or entering them in a directory of objects known at that site. Once this is done, the incoming reference may be replaced by a much shorter (e.g., 8-16 bit) reference which is unique within the processor, and which identifies the directory entry containing the full information about the object. Incoming object texts may thus be converted to the much shorter internal form, and the full inter-processor form regenerated from the internal form when a message must be sent. Obviously, the details of this translation can vary between sites, depending on various attributes of the host machines (processor architecture, memory size, word size), as long as some standard protocol for

communicating with neighbors can be maintained.

Similarly, details of the internal format of object texts may vary from one processor to another, but one or more external standards will have to exist for communication between processors. For concreteness, we describe briefly a set of formats used by computer simulations written as part of this research.

For numbers, there is no text. Instead, a short form of object reference is used, containing the bits of the number and a tag identifying the object as a number. ASCII strings are also supported. Here, the text contains the bytes of the string, followed by a zero byte (but no references to any other objects).

Mu-expressions are represented using a complicated format which indicates the number and identities of the free and bound variables of the mu-expression, followed by the number of events in the body and references to the components of those events. Since these components are other objects, the text of a mu-expression may actually contain references to other objects (for example, other mu-expressions).

A reasonable mu-calculus interpreter is likely to work by creating closures rather than by substitution, so an object type for the closure of a mu-expression (let us call this an "actor") is needed. The text of an actor is just a sequence of object references. The first is a reference to the mu-expression being closed, the remainder a list of references to the values of free variables in the mu-expression. The order of these values in this list is determined by the representation of the mu-expression.

The representation of tokens involves three kinds of objects: read sides, write sides, and token bodies. The text of a read or write side simply contains a reference to the corresponding token body. The text of the body is organized as a table somewhat like Figure 2.7, containing information about objects sent to the

read and write sides. Since the text of a token body can change, multiple copies cannot be made, as with all the kinds of objects discussed up to now. However, the text of a token body can be partitioned among several sites, in such a way that any individual entry appears at only one site. Adding a new entry then requires notification of all sites containing any part of the text so that the proper interactions can occur—a mechanism by which this can be accomplished will be described later.

The representation of cells is similar to that of tokens in that there are three kinds of objects involved: update ports, contents ports, and cell bodies. As in the case of tokens, the text of an update or contents port simply contains a reference to the relevant cell body. A simple scheme for representing cell bodies is one in which the text of a cell body is just a reference to the object which is the current contents of that cell. Since this text is, of course, mutable, care must be taken that there be only one copy of it at any time. Thus if activities on several different processors are all accessing the same cell body, a certain amount of communication overhead will be incurred. Ways of ameliorating this situation will be discussed later.

The representation of semaphores could obviously be similar to the representation of cells, with additional mechanism for handling waiting events. The details of this do not shed much additional light on the fundamental concepts of this chapter and will not be discussed further.

3.2.2: Dynamics of the System

The static and structural aspects of our implementation have now been described in sufficient detail that we can proceed to consider its dynamics, that is, the motivation and mechanism surrounding the scheduling of events, sending of messages, and so forth. In the remainder of this chapter, we shall generally be concerned with a "steady-state" situation in which, as a result of some unspecified history, several processors in the system have been assigned things to do, and need to communicate with other processors containing, for whatever reason, data upon which they need to operate. In this section, therefore, we shall concentrate on developing some intuition for such a "steady-state" situation, briefly exploring mechanisms by which it might come to be and strategies by which it might be managed.

Let us assume that initially, by some means such as an operator typing at a keyboard, one processor somewhere in the network has been given an event to process. As we have seen, each processor maintains an event list, so this processor will now have one event on its event list. The goal of each processor is to empty its event list, so our processor will take the new event off its list and see what to do with it. Most likely, the event will cause some computation to occur and then result in a new event's being added to the event list, whereupon the whole cycle will repeat. As long as this situation persists, and each event causes exactly one new one to take its place, there is little opportunity for other processors to get into the act.

Let us suppose therefore that at some point an event causes two or more events to be added to the event list. Henceforth there will be several events vying for our processor's attention at the same time. The processqr might continue to operate on all the events itself by always taking the oldest event from the

event list, producing its consequent events, and placing these at the end of the event list. To reduce queuing and dequeuing overhead, it might be smarter to go through several generations of consequents from an event before putting the new event(s) back on the event list. Another way to speed things up is to take advantage of the other processors in the system. If there are enough events on the event list, the overhead of sending some of them to a neighbor should be worth the increased processing power thus brought to bear on the problem.

The process by which it is decided what events to execute where is the subject of a subsequent section; before taking that up it is a good idea to have a short look at the mechanics of moving events (and, as a consequence, objects) around. Sending an event to another processor is simple enough—all that is required is to send a list of references to the objects participating in the event. This, however, represents only a small part of the overhead required to actually bring that processor into the action. Before the new processor can make any sense at all out of the event, it will need a copy of the text of the receiving object (the first object) of the event. If it does not happen to already have this text, it will have to send an inquiry for it to some processor which does. (This will probably be the original sender of the event; however, it may be that no neighbor of the inquirer has a copy of the text which enables it to reply immediately to the inquiry. In this case, the inquiry must be forwarded until it reaches a processor capable of replying, whereupon the reply must be forwarded back to the original inquirer.) During the period that this inquiry is being sent and replied to, the event cannot be processed. Depending on the nature of the receiving object, further inquiry-response cycles may be required to gather enough information to process the event. Finally, when the event is processed, a new event or events will most likely be generated, containing references related to those present in the original.

Frequently, the texts of these objects will not be available locally either, so additional inquiry-response delays may ensue as subsequent related events are processed. Thus the true overhead associated with sending an event to another processor is the overhead required to establish a "working set" for that event and its consequents on that processor.

Having exposed the drawbacks of sending an event to another processor, it is worth mentioning some mitigating factors. First, if the event really represents the beginning of a computation that will proceed for a while without requiring too much communication with other activities on the same processor, using another processor can still come very close to doubling the effective computing speed of the system. Second, the efficiency of the system can probably be improved greatly through various forms of tuning. For example, the sender of an event might also send a selection of object texts likely to be asked for by the receiver in the process of setting up its "working set." Similarly, the reply to an inquiry might not be just one object text, but a collection of related texts including the one asked for. These strategies would not do much to reduce the number of bytes sent over communication lines, but would reduce the number of wasteful delays between sending of inquiries and receipt of responses.

Fortunately, the working set that must be accumulated by an event running on a new processor can often be reasonably compact. Suppose, for instance, that the event is a function receiving arguments and a continuation. A working set that will allow this computation to proceed for a while would include the body of the function (or at least those parts of the body which will be exercised by the arguments given) and the texts of the arguments. If the arguments are numbers, no extra information at all is required. If the arguments are highly structured, it is probable (depending on the nature of the function) that only a portion of the texts

accessible from the argument references will ever be accessed. Note that the text of the continuation is not needed at all—at least not until the function returns a value to its continuation, which might be a natural point to transfer responsibility back to the first processor, where the text of the continuation is presumably located.

To summarize, then, we can imagine the system beginning to operate with one event at some particular processor. Activity will spread to other processors as the parallelism of the program increases and overloaded processors send events to their less-loaded neighbors. Perhaps at some later point the various threads of the computation will either die out or begin to come together again, perhaps using tokens to synchronize. Thus the number of active processors might shrink until there is again only one active processor with one event to process—perhaps "print out the answer." Of course, another possibility is that the system is constantly receiving new events to process as users come up with new tasks for the system to perform. This will lead to more of a steady-state situation where, at any time, some tasks are just beginning, some just ending, and others in various growing or shrinking phases. In either case, the system will suffer from some amount of overhead and inefficiency as processors build up working sets for newly acquired events, but hopefully will gain even more from the application of extra processing power to its business.

3.3: Object Management

As implied by the suggestions made in a previous section regarding the representation of objects internal to processors, it is not intended that every processor should "know about" (i.e., contain references to) *every* object in the system. In fact, it is not intended that *any* processor should need to know about every object in the system. These two constraints complicate the problem of object management in our system. Fundamentally, we desire that local changes in the status of an object should require only local processing. Thus if a processor passes a reference to some object *X* to a neighbor which did not previously contain any references to *X*, a local adjustment to the data base about *X*, preferably involving only the two processors carrying out the transaction, should suffice. A strategy requiring the notification of all processors with references to *X* that a new member had joined their club, for example, is unacceptable.

This point seems fairly trivial, but a related point, also concerning object management, is less so. So far, we have discussed the significance of the set of processors which contain references to an object. Some subset of this set have a special responsibility with regard to the object, however; they are the custodians of copies of the text of the object. (Objects without texts, such as numbers, are exempt from all the considerations discussed in this section—if the reference contains all the information about an object, no fancy bookkeeping is required since the reference will obviously be present wherever the object is referenced!)

The special responsibility imposed on a processor having a copy of a text is twofold: first, the processor must cooperate with all other custodians of texts for that object to insure that at least one copy of the text exists at all times; second, the processor may be asked to respond to inquiries from other processors not having copies of that text. Viewed from the other side, a processor with a

reference to an object but no text must know where to send an inquiry if it becomes necessary to obtain a copy of the text. Several solutions to this problem can be imagined.

A solution that can be rejected almost immediately is to have one designated processor be the custodian of all object texts in the entire system. Not only would this be a major bottleneck and reliability problem, it is conceivable that no single node in the network would have sufficient storage to hold all those texts.

The next solution, and a fairly plausible one, is to associate with each object a designated processor which is to be the custodian of its text, perhaps the processor where the object was created. This answers the objections raised to the previous scheme, leaving as the only technical problem the matter of properly routing the inquiries and replies. This problem, however, has been solved in many contexts (e.g., the ARPANET[21]). This scheme is still open to objections, though. For one, a processor may send to the designated custodian for a text which in fact is also present at a much nearer processor. There is no mechanism for maintaining information which might help determine the most appropriate target for an inquiry. This might not be a serious problem on, say, a ring or Ethernet, where all processors are approximately the same "distance" apart, but becomes a more serious consideration if long-distance messages must be forwarded by many processors.

Another objection is that the custodian of an object cannot easily be changed. In violation of our desiderata stated above, if primary custody of a text passes even from a processor to its immediate neighbor, all possessors of references to the corresponding object must be notified. Failing that, a "forwarding address" must be left with the old custodian, leading to extra delay in responding to inquiries, and imposing an extra burden on the old custodian that may nullify some of the economies achievable by transferring custody in the first place. The scheme

to be presented below does incorporate the notion of custody, but many processors can serve as custodians for the same object at the same time, and all custodians are in principle equal. Furthermore, although forwarding information does need to exist, it must only be kept by processors actively concerned with an object; no permanent obligation rests with, for example, past custodians for the object.

To summarize, then, we envision a scheme in which at any time some subset of the processors in the system may possess references to a particular object, and some subset of those, in turn, will be custodians of copies of its text. Ideally, this situation should be completely fluid; only those processors which need (for purposes of their own internal operation) to have references to an object ought to be forced to keep them, and processors with no further need to have the text of an object ought not to be required to hang onto it. This should happen without regard to the ancient history of the object (i.e., the creator of an object ought not to be forced to accept any special responsibility for its continued existence). Finally, a local change in the set of processors knowing about or having copies of the text of an object should have only local ramifications. The scheme to be presented below satisfies several of these criteria to a greater extent than the schemes discussed so far. Nevertheless, it still does not have all of the desired characteristics, and there is clearly room for further improvement.

3.3.1: Reference Trees

The object management algorithm developed in this research involves maintaining the processors which contain references to an object in a *connected, acyclic* graph called the *reference tree* for that object. Each reference tree consists of some subset of the nodes and arcs (processors and inter-neighbor links) of the network. The nodes which belong to the reference tree are chosen to be those

processors in the network which contain references to the object, and the arcs are chosen in such a way that (1) the reference tree is *connected*, i.e., it is possible to reach any node in the tree from any other node, traveling only over arcs that are in the tree, and (2) the tree is *acyclic* in that the arcs in the tree form no closed loops. (Note that the arcs in the reference tree are undirected, hence requirement (2) means that there should be no undirected cycles.) Put another way, there is a unique path (using only arcs in the tree) from any node in a reference tree to any other. One additional consideration which is not obvious from the above description is that every arc in a reference tree goes between two nodes that are in the tree—no reference tree can include arcs between two nodes not in the tree, or even between one node in the tree and one node not in it. Reference trees are so named because they form unrooted trees embedded in the network (see Figures 3.2 and 3.3).

It is important to note that, in general, the reference trees for different objects need bear no relation to each other. In particular, it is not the case that there is one "reference tree" in the network, used for all objects. Central to the concept of reference trees is that they are free to grow and shrink dynamically, following changes in the roles of the corresponding objects in the operation of the system.

Also significant is the fact that reference trees can be maintained by a completely distributed mechanism in which each processor in a tree remembers only the state of its immediately adjacent links (i.e., whether each link is in the tree or not). Processors not in the tree for an object, of course, have no references to the object, and need remember no information about it. Even the cycle-free nature of the tree can be preserved on the same strictly distributed basis—no central clear-

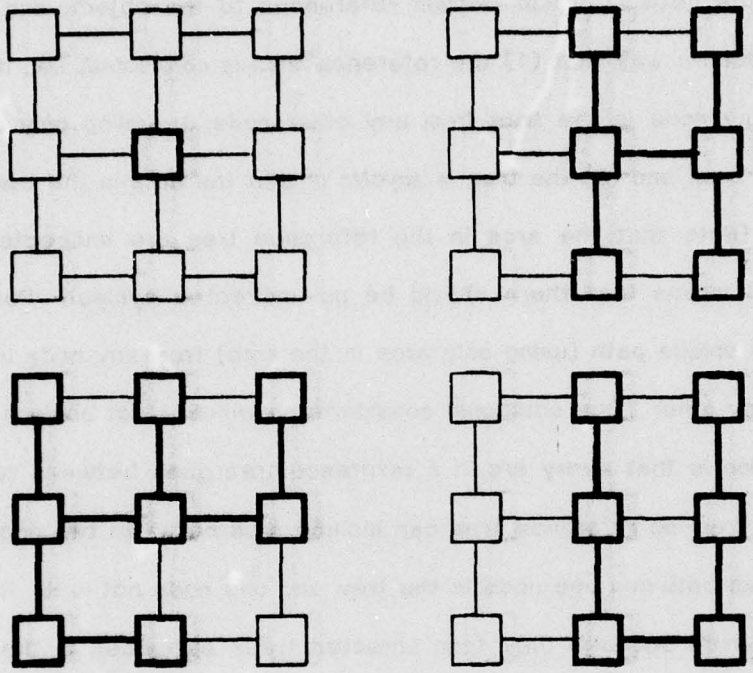


Figure 3.2: Examples of reference trees (in heavy lines)

inghouse is needed to determine whether a cycle is being formed.

As mentioned previously, some subset of the processors having references to an object will also be custodians of a copy of the text of the object. In our current scheme, this means that some subset of the nodes in any reference tree will in addition fill the special role of custodians. In fact, the primary reason for requiring reference trees to be connected is so that any processor with a reference to an object can always communicate with a custodian for that object simply by following links that are part of its reference tree. Indeed, we will require that *all communication concerning an object travel strictly over links that are in the reference tree for that object.* The only exception to this rule occurs when a new node is in the process of being added to the tree; this case will be discussed in

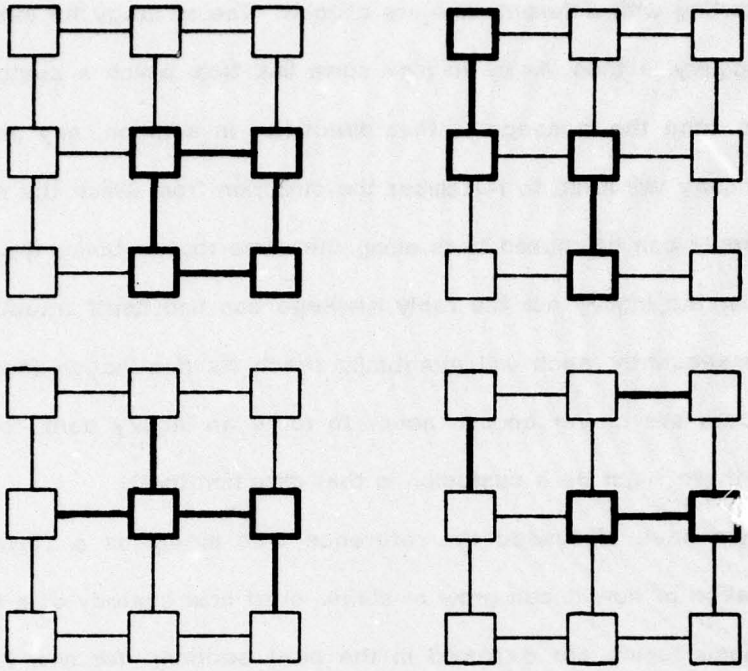


Figure 3.3: Examples that are not reference trees

more detail in the next section.

Given that all requests for the text of an object will travel through the reference tree for that object, and given that, if routed properly through the tree, such requests are guaranteed to eventually reach a custodian of the text, the question remains of how to make sure such a request is indeed routed properly. It is here that the acyclic nature of the tree comes into play. For every adjacent link that a processor believes to be part of the reference tree for some object, that processor will maintain an additional piece of information indicating whether any processor reachable through that tree starting with that link is a custodian for the text of that object. Since the tree is connected, it is guaranteed that if a custodian exists, it is reachable from any node in the reference tree starting from some link. Since the tree has no undirected cycles, it is certain that the sets of processors

reachable starting with different links are disjoint. The strategy for initiating or forwarding an inquiry is thus simply to pick some link from which a custodian can be reached, and send the message in that direction. In addition, any processor forwarding an inquiry will need to remember the direction from which the inquiry came, so that the reply can be routed back along the same route. Since the tree has no cycles, neither the inquiry nor the reply message can find itself traveling around a loop, and consequently each will eventually reach its destination (assuming message forwarders are clever enough never to route an inquiry back to its sender, even though there might be a custodian in that direction too!).

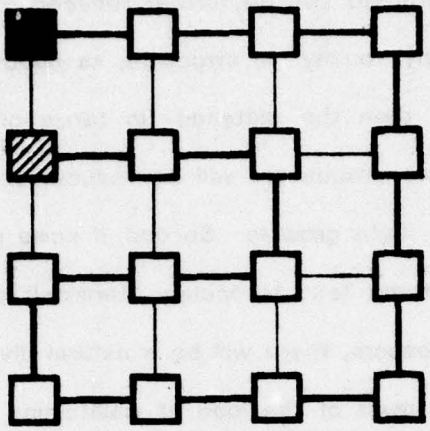
So far, we have discussed the reference tree simply as a static structure, with no indication of how it can grow or shrink, or of how custody of a text may be changed. These topics are explored in the next section; for now it suffices to note that, in accordance with our desiderata, local changes require only local modifications to the distributed data base which serves to represent the tree. Even this scheme, however, still falls short of our goals in several ways.

One immediately obvious liability of the reference tree approach arises from the requirement that the tree remain connected. This requirement means that if two far-apart processors both need to maintain a reference to an object, several processors between them (a sufficient number to keep the two ends connected) must also be part of the tree, even though they have no other involvement with the object. The impact of this is not tremendous, for the amount of overhead involved in membership in a reference tree is not large. In any scheme, presumably, the several processors between would be involved in forwarding messages from one end to the other, so the only extra overhead imposed by the reference tree mechanism (assuming our cellular network as the underlying hardware) is the

static storage overhead of remaining part of the reference tree.

The impact of this overhead can be further reduced by several mechanisms. First, if the network is fairly "bushy" in structure, as opposed to, for example, a long chain of processors, then the distance (in terms of the number of links traversed) between any two processors will be reduced, cutting the length that such long reference chains might grow to. Second, if some processors tend to run along "trunk lines" while others tend to occupy "terminal" positions connected to only one or two other processors, there will be a natural division of labor with the "trunk" processors bearing most of the load of maintaining long reference trees. Such processors might be specifically designed with this service role in mind. Third, some event distribution strategies will tend to prevent such long reference chains from developing. Finally, it is possible under some circumstances to remove the requirement that a reference tree remain connected–this will be discussed in Section 3.3.5.

The other restriction that applies to reference trees is that they must be kept free of cycles. This is not too difficult–the method will be discussed below. The prohibition of cycles causes some other problems, though, related to the fact that even though two neighbors are part of a reference tree, the link between them may not be (if its addition to the tree would close a cycle). We have required that all communications concerning an object travel strictly along the links in its reference tree. Thus a processor might, for example, issue an inquiry along some link in the tree; that inquiry might travel a long distance before it could be satisfied. It is possible, however, that sending the inquiry over some link *not* in the tree would result in its reaching a neighbor *in* the tree, and that the request could be satisfied immediately by that neighbor (see Figure 3.4). This is just one illustration of the various ways in which a reference tree can fail to be optimal; the question of how



The solid box represents a processor with a copy of the object's text; the shaded box represents a processor inquiring for the text.

Figure 3.4: A non-optimal reference tree

to reorganize the tree to a better shape will be discussed later.

3.3.2: Reference Tree Maintenance

This section describes an interprocessor communication protocol which may be used to grow and shrink reference trees while preserving the required connectedness and freedom from cycles. Also described is another protocol, fairly independent of the first, which may be used for communicating and managing custody of object texts. These protocols turn out to be fairly intricate and inelegant. However, they do work (further justification of this claim and the story of how these protocols were developed are presented in Appendix A). The protocols grew to their present level of complexity due to the propensity of multiprocessor systems to deadlock and/or reach inconsistent states through unfortunate timing of independently initiated requests. While it is hard to believe that there do not exist simpler protocols which will do the job, several simpler protocols were considered and

rejected in the process of developing these. In the discussion below, we explore the reasons for the failure of these simpler approaches and the motivation for our current scheme. One final note: the protocols discussed here make no attempt to recover from damaged or lost messages, or messages arriving out of order. These problems can be solved by various well-known means[21] which may be assumed to provide the underlying protocol on each link. It may be that minor modifications of the concepts to be presented would eliminate the necessity for any such underlying protocol, but such work would further complicate the algorithms presented below and was judged to be beyond the scope of this research.

3.3.2.1: Changes in Reference Tree Membership

This protocol is concerned with maintaining that part of the reference tree data base devoted to recording whether or not particular processors or links are members. Custody issues play no direct role here, although changes in custody will often be the cause of various transactions of the kinds described below.

The membership protocol involves six basic kinds of messages, whose meaning is roughly as given in Table 3.5. Each message is, of course, specialized by identification of the object (and hence reference tree) that it pertains to.

Message	*Meaning*
R+	object reference plus request to add link
R-	object reference without request to add link
+	add link to tree
-	request to drop link from tree
A+	positive acknowledgment
A-	negative acknowledgment

Table 3.5: Membership protocol message types

These message types, of course, derive their exact meaning from the way in which

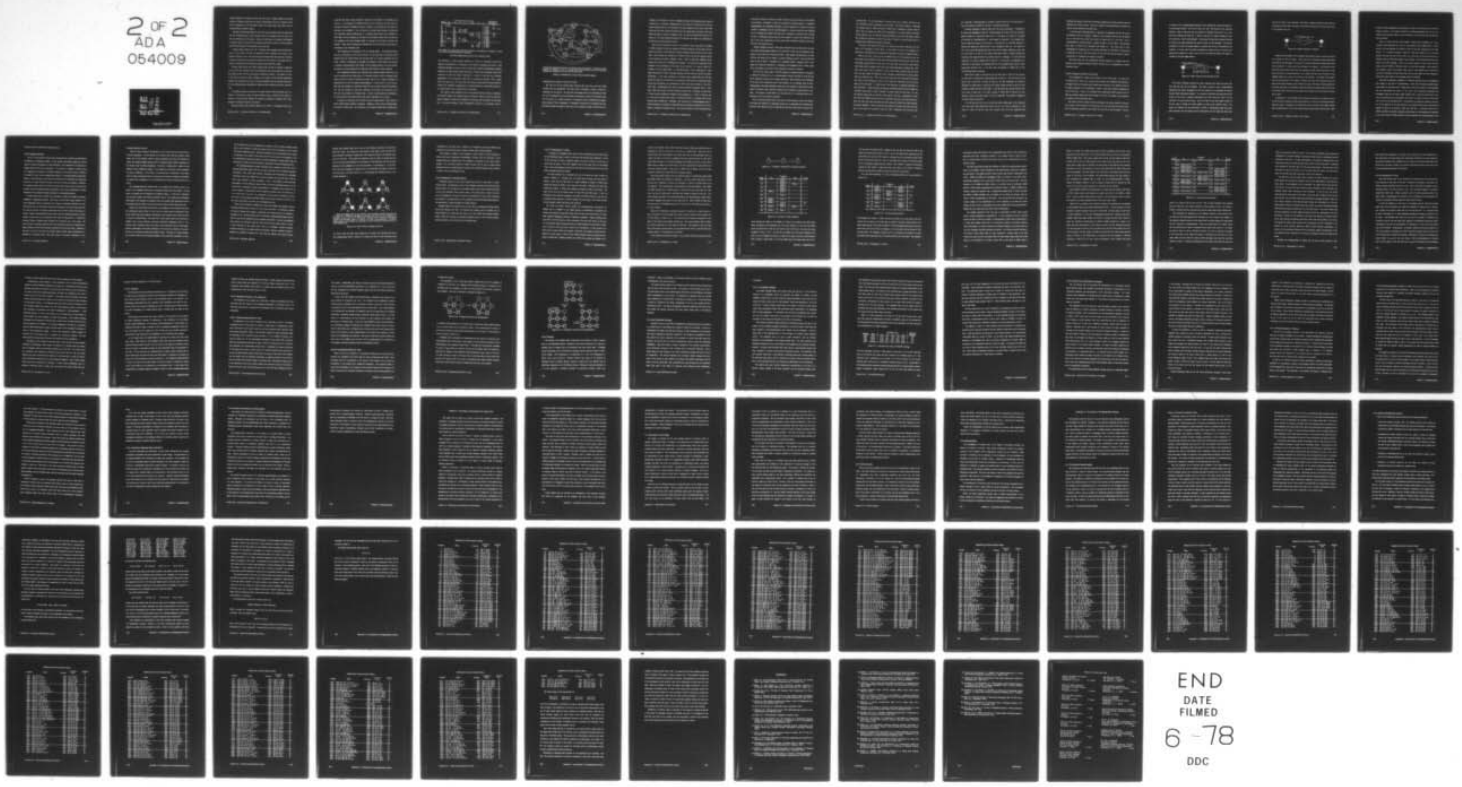
they are used by the protocol, but a few general comments are in order regarding the circumstances under which the messages may be sent. A+ and A- messages are sent only in response to other messages. + and - messages are sometimes sent spontaneously, and are sometimes sent in response to R+ messages. R+ and R- messages are always sent spontaneously; that is, in response to some external stimulus, such as the need to send a text containing a reference to the object, rather than in response to any of the messages listed in Table 3.5. R- is the only kind of message that never provokes a response, and corresponds to communicating a reference to the object over a link which has at least provisionally been included in the reference tree for the object. R- messages cannot be sent under all circumstances, however; sometimes an R+ message is required to establish the intent to add the link in question to the reference tree.

The membership protocol operates by associating with each end of each link a *state* for each possible object. In terms of implementation, each processor must maintain a data base for each object it has a reference to, indicating the state (with respect to that object) of each link adjacent to the processor. It is important to realize that the two processors at the ends of a link may have different ideas of the state of the link; this may be as the result of some intentionally introduced asymmetries discussed below, or it may occur if messages regarding the object have been sent at one end of the link but not yet received at the other.

The possible states may be grossly characterized as being either *stable* or *transient*. Stable states are states which might be expected to persist over a relatively long period of time. Transient states are those in which a message has been sent across the link and a reply is expected; the reply will cause a transition to some other state, either stable or transient. Of the two kinds of states, the stable states are the more important; the transient states exist to provide the

AD-A054 009 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 17/2
MULTIPLE-PROCESSOR IMPLEMENTATIONS OF MESSAGE-PASSING SYSTEMS.(U)
APR 78 R H HALSTEAD N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-198 NL

2 OF 2
ADA
054009



END
DATE FILMED
6-78
DDC

proper sequence of events so that the next pair of stable states to be established is consistent and does not result in partitioning the tree or closing a cycle. For the purposes of the discussion below, the states have been given one- to three-character mnemonic names.

Perhaps the state most likely to occur is X, which indicates that not only is the link not considered part of the reference tree, the processor does not even contain any references to the object. Clearly if a processor is in state X with respect to a given object for one link, the processor should be in state X (or the transient state X?) with respect to that object over every link.

A state closely related to X is N. In state N, the processor does contain references to the object, but does not believe the link in question to be part of the object's reference tree. This state may come about either because the processor is the only processor to contain any references to the object, or because the processor is connected to the reference tree by some other link or links.

Another stable state is M, which indicates that the link in question *is* believed to be part of the reference tree, and furthermore that this processor is currently the master of that link (for transactions involving that object). The master of a link is the only one that can effect changes in the status of the link—this asymmetry seems to be necessary to prevent confusion resulting from, for instance, both ends of a link simultaneously attempting to terminate their connection with the reference tree.

In a stable condition, the state at the other end of a link from M may be S, for "slave." A processor in state S cannot directly cause a change in the status of the link; it may however request the master to commence a change, and it may respond to changes ordered by the master.

The final stable state, closely related to S, is SR. A processor will go into

state SR from state S upon sending a reference to the object (R- message) over the link. It is necessary to remember whether such a reference has been sent by the slave because it modifies the proper response to an attempt by the master to terminate the connection. This is because the master might attempt to terminate the connection before receiving the R- message (which caused the transition to state SR), and the response by the slave to this attempt should take into account the possibility that that message might be received at the other end after the attempt. Under such circumstances, allowing the link to be broken could cause a partitioning of the reference tree.

This completes our discussion of the five stable states. The transient states will not be described to this level of detail. For the most part, they acquire their meaning from the stable states that precede them or to which transitions can be made. Instead of attempting to describe the meaning of these states, we present a complete state-transition table (Table 3.6) and diagram (Figure 3.7), and summarize below the normal sequences for effecting various kinds of state changes.

The fundamental principle that motivates this protocol design (other than the need to maintain the link data base in a consistent state) is that a processor must always be able to send a reference (of either the R+ or R- variety, whichever is appropriate) over any link *without prearrangement*. In other words, it is not acceptable that the sending of a reference should be part of some transaction beginning, say, with the sending of some other message, and allowing the reference to be sent only upon receipt of a suitable reply. In order to understand this requirement, we must examine the circumstances under which references may be sent.

In general, a reference will be sent as part of some event or text which is being communicated between processors. Sending a text involves communicating two kinds of references: the reference to the object whose text is being sent,

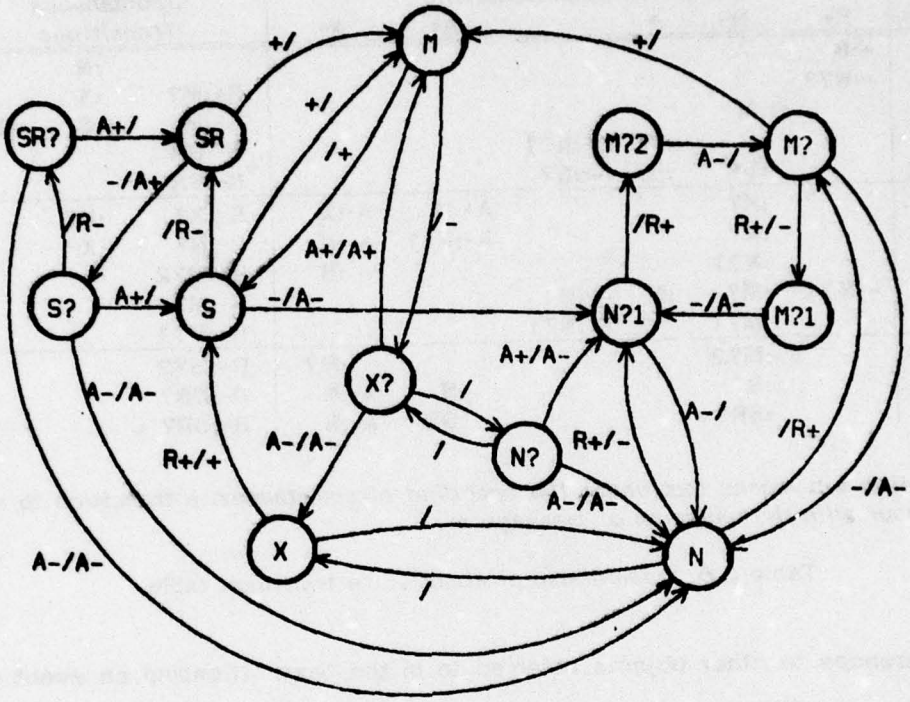
State	Transition Upon Receiving R+	R-	+	-	A+	A-	Spontaneous Transitions
X	+:S						:N
N	-:N?1						R+:M? :X
M		:M					R-:M +:S -:X?
S		:S	:M	A-:N?1			R-:SR
SR		:SR	:M	A+:S?			R-:SR
X?		:X?			A+:M	A-:X	R-:X? :N?
N?		:N?			A-:N?1	A-:N	R-:N? :X?
N?1		:N?1				:N	R+:M?2
M?	-:M?1	:M?	:M	A-:N			R-:M?
M?1		:M?1		A-:N?1			R-:M?1
M?2		:M?2				:M?	R-:M?2
S?		:S?			:S	A-:N	R-:SR?
SR?		:SR?			:SR	A-:N	R-:SR?

The notation a:b *means that under the specified circumstances, a transition to state* b *can occur with the emission of message* a.

Table 3.6: Membership protocol state transition table

and references to other objects referred to in the text. (Sending an event only requires communication of references to the objects composing the event.) Thus obtaining clearance to send a text or event may involve simultaneously obtaining clearance to send several object references. Unless every processor always has clearance to send any object reference, it is easy to see how the piecemeal aggregation of such clearance could lead to a deadlock on the link. This is especially true when, as is the case with this protocol, transactions involving different objects are completely independent—no overall master-slave relationship applies to all communication over a particular link, for example.

This need to avoid deadlock is one of the primary factors acting to complicate the protocol design, and requires that any processor always be able to send any object reference without the possibility of confusing the processor at the other side. The only exception to this requirement is state X—if a processor has no



Transitions caused by receiving R- messages have been omitted. Transitions accompanying the sending of an R- message have been omitted where the state after sending the message is the same as the state before.

Figure 3.7: Membership protocol state transition diagram

references to an object, it has none to send!

We now turn to the various state changes that may occur, and why those changes may be appropriate. We start with a processor in state X, having no references to the object. The only kind of message that can be received in state X is an R+ message from some processor attempting to extend the reference tree for the object as a result, perhaps, of sending an event mentioning the object. Upon receipt of the R+ message, a + message is returned as an indication that the link should indeed be added to the tree, and a transition is made to state S in anti-

cipation of the sender of the R+ message entering the M (master) state when it receives the + message. Simultaneously, the states of all other links to the processor change from X to N, indicating that the processor is now part of the reference tree, but that it does not believe any of its other links to be part of this tree. (Also, any links in state X? change to N?.) As may be noticed from Table 3.6 or Figure 3.7, the state change between X and N does not require any notification of the party at the other end of the link.

Now that our processor is part of the reference tree, it may need to attempt to further extend the tree by sending a reference along one of the links just converted to state N. From state N an object reference must be sent as an R+ message. Upon sending the message, the sender's state for that link changes from N to the transient state M?, awaiting a reply. While in state M?, additional references may be sent as R- messages if necessary. The reply to R+ depends on the condition of the processor at the other end of the link. If it was in state X, it changes to S and replies with +, as described above. Upon receiving the + message, the sender of the R+ message change from M? to M, and the link has been established. If the other processor is in state N (also possibly M?) then the link cannot be added to the reference tree because it would close an undirected cycle (since both processors are connected by some other route already in the reference tree). Consequently, the other processor responds negatively, with a - message. When the sender of the R+ message receives the - message, it moves back to state N while emitting the negative acknowledgment A-. When it receives the A-, the other processor then returns to state N from N?1, a transient state it had entered after sending the - message. The extra level of acknowledgment here is needed because a processor in state M? may send object references as R- messages, a capability it needs to have. The other processor must be prevented from

returning to state N, or worse, to state X, before its reply is known to the originating processor; otherwise, it would be confused by receiving these R- messages. Consequently, the responding processor is held in state N?1, in which it *is* able to accept R- messages, until the acknowledgment A- arrives, certain to have followed any R- messages that might have been sent. Processors in states N and X cannot accept R- messages because they are not requests to extend the reference tree, and if they were interpreted as such, could cause confusion resulting in the partitioning of the reference tree.

Another possible scenario is that two processors, both in state N (for the same link) might simultaneously attempt to add that link to the tree by sending R+ messages to each other and entering state M?. Under these circumstances, it is clear that the link should not be added, to prevent forming a cycle. Thus each M? will reply to the R+ with a - message and a transition to M?1. Receipt of the - messages will prompt the sending of A- messages and transitions to N?1. Finally, when the A- messages are received, both processors will revert to state N.

M?2, the only other state in the M? complex, is required because a processor in state N?1 (waiting for an A- acknowledgment before returning to state N) may find it necessary to send out an object reference. Since the link is not considered to be part of the reference tree at this moment, an R+ message must be sent. Consequently, state M? (awaiting a response to the request to add the link to that object's reference tree) should be entered after receiving the expected A- message. The function of state M?2 is to wait until the A- is received, since an A- message cannot be handled in state M?.

Once it has been agreed that a link is part of the reference tree for an object and things have settled to a quiescent state (i.e., no messages are in transit), one processor (the master) will be in state M and the other (the slave) in state S (or

possibly SR). It is a simple matter to reverse the roles of master and slave, but the transaction must be initiated by the master. The master sends a + message and enters state S. When the slave receives the + message, it enters state M. Since only the master can initiate this transaction, if a slave wishes to become master (which it may for reasons discussed below) it must first use some mechanism outside this protocol to induce the master to begin the role-reversal (such as sending some "request-for-mastery" message).

Of course, both master and slave can freely send object references (in the form of R- messages) to each other. In the case of a slave, the sending of a reference is accompanied by a transition to state SR, for reasons explained below.

Having seen how a link may be established to be in the reference tree, we now come to the question of how a link may be deleted from the tree. Due to the connected, acyclic nature to the tree, it is true that, as we have seen above, every time a link is added to the tree a new node (processor) is added also. Similarly, every time a link is deleted, a node is also being removed from the tree. Thus the only reason for deleting a link is because a node wants to remove itself from the reference tree. This in turn will be caused by that processor's discovery that none of the references it has to the object are reachable from any active data on that processor. In other words, the reference is garbage-collectable on that processor. A later discussion of garbage collection will make more precise the conditions under which a node is allowed to remove itself from the tree; for now, we observe simply that it must be a leaf node of the tree. Equivalently, it must have only one neighbor also in the tree. Otherwise, removal of the node would partition the tree, since the node forms the only connection (within the tree) between its various neighbors. Thus a processor may attempt to remove itself from the tree only if all its links but one are in state N (or N?). Additionally, that one link must

be in state M; if the processor is currently a slave on that link (for that object), it must first induce the master of the link to relinquish its mastery.

A master requests to remove itself from the tree by sending a - message to its slave and changing to state X? (simultaneously all N links from that processor should change to X and all N? links to X?). If the slave is in state S, it accepts the deletion by responding with A- and changing to state N?1; the standard link-refusal sequence takes over from there. If the slave is in state SR, however, it refuses the deletion by replying with A+ and changing to state S?. The reason for this is that state SR indicates the slave has sent an object reference to the master, and there is no way (in this simple finite-state model) of telling whether the reference was received before the master sent the - message. If it was received afterward, the master, though still in state X?, is once again in possession of a reference to the object, and allowing the link to be deleted might result in partitioning the reference tree. To be safe, then, the deletion is refused and a short hand-shaking phase is entered.

When the A+ reply is received by the old master still in state X?, the old master will return to state M and acknowledge with another A+. This will cause the old slave, now in state S?, to return to state S. By this mechanism, its record is cleansed of having been in state SR, and unless further activity occurs the next attempt to delete the link will succeed. If the old slave sent another object reference while in state S?, it will have changed to state SR? to record that fact. Receipt of the A+ will then cause a return to state SR. State SR? is needed for the same reasons that SR is.

It is possible that the old master will have been added again to the reference tree (over some other link) before either the A+ or A- reply is received. In this case, its state for this link will have changed from X? to N?, indicating that

although the status on this link is still being negotiated, the final outcome must be negative (to prevent cycles). There is no danger of partitioning the tree since this node is now connected by another path.

If the old slave responded with A-, signifying its agreement that the link should be deleted, there is no problem. However, if the response was A+, the old master now in state N? must quash the attempt to keep the link in the tree. This is accomplished by answering with A- instead of with A+, as X? would. The old slave then goes to state N and responds with another A-. This extra level of acknowledgment is necessary, as in the earlier discussion of state N?1, to insure that a reference in the R- form can be sent at any time from the old slave, up until it receives the A- message and changes to state N.

We have seen how the membership protocol operates to maintain the connected, acyclic nature of reference trees. We now turn to a mechanism for managing object texts.

3.3.2.2: Changes in Object Text Custody

The management of object text custody has two basic goals: to insure that no object text is "lost," (i.e., to insure that at least one processor has custody of an object's text at all times), and to keep each processor in the reference tree for an object apprised of the directions in which it may send inquiries requesting a copy of the text. There is an additional issue regarding mutable objects such as cells and tokens--making sure that changes in the text are visible to the appropriate agents at the appropriate times.

We discuss first the means by which processors can obtain copies of the text of an object. Each processor in the reference tree for the object must keep three bits of information specific to that object for each link to that processor. (This is

in addition to the record-keeping required for the membership protocol described in the preceding section.) The first of these bits (the "text-this-way" bit) indicates whether a copy of the text may be reached by following some path in the reference tree starting with that link. If this bit is not set, it is clearly fruitless to send an inquiry for a text in this direction. The second of these bits indicates whether any inquiry has been received over that link (and not yet satisfied). The inquiry-received bit is used for routing replies to inquiries that had to be forwarded to be satisfied. It is also used to prevent forwarding an inquiry back to its sender even though there might be a copy of the text in that direction—Figure 3.8 shows an example of how this might happen.

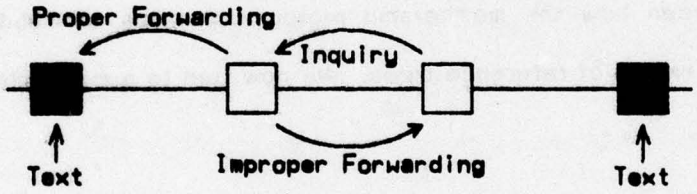


Figure 3.8: Inquiry received from direction of text

The third bit indicates whether any inquiry regarding that object has been sent over that link and not yet satisfied. This might be used to aid in retransmitting apparently lost inquiry messages, but its real purpose is to streamline the inquiry process while preventing deadlock. If an inquiry is received over some link and an inquiry for the text is already outstanding over some other link, then it is not necessary to send another inquiry; when the reply to the first inquiry arrives, it may be used to satisfy the latest inquirer as well as any previous ones. If an inquiry is received over the same link to which an outstanding inquiry has been sent, however, the new inquiry must be forwarded, and over some link other than

the one on which it was received. Otherwise a deadlock situation could result as in Figure 3.9, where each processor could decide to ignore the other's inquiry since it had already sent one.

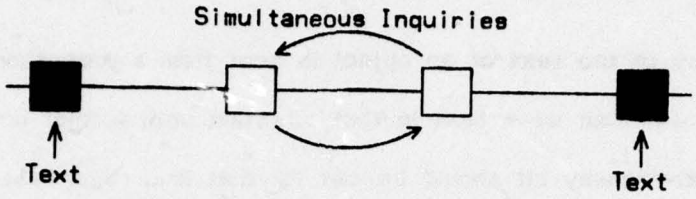


Figure 3.9: Possible deadlock in inquiries

There are two reasons for a processor that does not have a copy of the text of an object to send an inquiry: either the text is needed by some computation occurring on that processor, or an inquiry for the text has been received from some other processor. In either case, the strategy is as follows. If an inquiry for that text has already been sent from this processor, simply await the reply to that inquiry (except as discussed above). If no inquiry has been sent, send one to any neighbor which has a copy of the text in his direction. Of course, if the inquiry must be sent because of an inquiry received, the inquiry sent should not be sent back in the same direction, even if there is a copy of the text in that direction (if an inquiry was sent in our direction, there will always be another link which is part of the path from the source of the inquiry to the copy of the text that the sender had in mind).

If custody of a text changes (for example, in response to an inquiry), a copy of the text will in general be sent along some link. As we have already discussed, this may result in changes to the reference trees not only of the object whose text is being moved, but of any objects directly referenced from the text. Addi-

tionally the bits pertaining to the location of texts in the reference tree must be updated properly. The inquiry-received and inquiry-outstanding bits are set and cleared at the obvious times, but the treatment of the text-this-way bit deserves closer attention.

When a copy of the text of an object is sent from a processor over a link, obviously there will then be a text in that direction until further notice. Consequently, the text-this-way bit should be set for that link, regardless of its state before. If a processor receives a copy of a text over some link, it is not clear whether this was the only copy of that text in that part of the reference tree, or whether there are still other copies in that direction. This information must be contained in the text message. Thus if a processor wishes to send a text and also keep a copy for itself, it should set the bit in the text message. If the sender is not keeping a copy, and no other copies exist in the sender's part of the reference tree (reachable via links other than the one over which the text message is being sent), then the bit in the message should be cleared.

This level of protocol is sufficient to correctly maintain the text management bits except in one set of circumstances: when two processors simultaneously send copies of the same text to each other, and one (or both) of the messages has its text-this-way bit clear. In this case, the processor(s) receiving the message(s) with the bit clear would "forget" that a text had been sent to the other processor(s), and consequently would have an incorrect picture of the location of texts in the reference tree. This difficulty is avoided by requiring a processor to be master of a link (with respect to the object whose text is being sent) before the text of an object is sent over that link. In practice, this is not a difficult condition to observe, although it does mean that occasionally the response to an inquiry must be further delayed until the responder can obtain mastery of the

link to be used for sending the requested text.

3.3.3: Garbage Collection

So far in this section we have been discussing the creation and maintenance of objects in a distributed system. In any system with limited resources, another aspect of object management is also important: the identification and disposal of objects that will never be used again. It is possible to imagine a system where the programmer is required to explicitly deallocate such objects—such a discipline is standard in many current programming systems. In message-passing computation, however, and especially in the style we should like to encourage as making the best use of a large distributed system, this can be quite inconvenient. (Consider, for example, being required to notify the system upon the last use of every actor created during a message-passing computation.) We are thus motivated to explore schemes for the automatic garbage collection of inaccessible objects.

To date, there have not been many attempts to solve the problem of garbage collection in distributed systems. One piece of work that seems related is by Peter Bishop[4], where he discusses the concept of garbage collection by *areas*. Records are kept of references across area boundaries, and these *inter-area links* are used to prevent any objects referenced only from other areas from being collected. Various researchers[12] have noticed that this garbage-collection scheme seems to have most of the properties that would be appropriate in a distributed garbage collector, defining each processor as containing one or more areas. We shall not look at garbage collection from this viewpoint, preferring to integrate garbage collection into the reference-tree approach we have been using. However, in the end, it will turn out that our scheme bears several deep resemblances to Bishop's—not surprising since both must share all functions essential to the

garbage-collection process.

When an object becomes inaccessible, it will in general have become known on several processors. In other words, its reference tree will have spread across some part of the system. None of these processors can take the initiative to delete the object outright because none in general knows whether references to the object exist on other processors. Therefore, it seems that it might be very difficult to ever reclaim the object. If the object is only known on one processor, the story is different. In this case, it is obvious that no references to the object exist on other processors (else the reference tree would be larger) and therefore the object can be deleted if it is not referenced on the one processor where it is known.

Our garbage-collection scheme works by shrinking the reference tree of an object to be collected until only one processor knows about the object. At that point, the object can be collected by traditional means. In order for a reference tree to shrink, nodes must remove themselves from it. The sequences of messages exchanged in the process of breaking a link have already been discussed, but the circumstances under which a link may be broken without harmful effect have not.

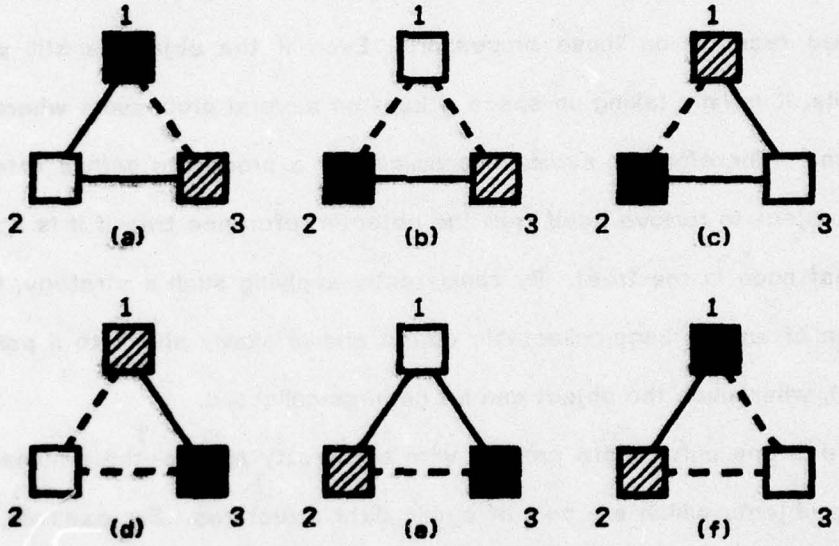
Clearly, any node which has more than one neighbor in the reference tree cannot unilaterally remove itself—if it did, the tree would become partitioned, since those nodes which were originally connected by the removed node would now have no means of communicating. Thus only "leaf" nodes—nodes which have exactly one neighbor also in the reference tree—may disconnect themselves from it. Fortunately, since reference trees are acyclic, every reference tree has leaf nodes. Another requirement is that some processor in the tree have custody of a copy of the text of an object as long as that object exists. Therefore, if a node attempting to remove itself from an object's reference tree is the sole custodian of a text

for the object (as can be determined by looking to see if there is another custodian in any direction), it must first preserve the text by passing it on to its neighbor before breaking the link. Since both sending a text and breaking a link require a processor to be master of the link, once the text is sent to the neighbor it can be considered preserved, provided the node attempting to delete itself does not relinquish its mastery of the link between sending the text and breaking the link.

The garbage-collection scheme presented here depends on the fact that a garbage-collectable object will not be used anywhere once it becomes garbage-collectable. Thus, after some interval, processors with references to a garbage-collectable object may guess that it can be collected by the fact that it has not been used recently on those processors. Even if the object is still potentially accessible, it is only taking up space if kept on several processors where it is not being used. Therefore, it seems economical for a processor with a reference to such an object to remove itself from the object's reference tree if it is able (i.e., if it is a leaf node in the tree). By consistently applying such a strategy, the reference tree of any garbage-collectable object should slowly shrink to a point (a single node), whereupon the object can be garbage-collected.

There is one unfortunate problem with this pretty picture—the problem involves collecting objects which are part of cyclic data structures. For example, consider an object A whose text contains a reference to B, whose text in turn contains a reference to A. Assume further that the structure is garbage-collectable—that neither A nor B can be reached from any ongoing computation. Then by the argument given above, the reference trees for both A and B should slowly shrink to a point. If both converge to the same point, there is no problem: ordinary garbage-collection techniques can easily handle the situation. However, another scenario is possible, as outlined in Figure 3.10. Here the two objects may spend forever

chasing each others' tails, and it may be that *neither* reference tree will ever shrink to a point. The reason this can happen is that when a text is moved from one processor to another it draws with it the reference trees for all objects referenced in that text. Thus when the reference tree for object A shrinks and the text of A moves from processor 1 to processor 2, the reference tree for B will be extended by the addition of a link from processor 1 to processor 2. If the reference tree for B attempts to contract next by the removal of processor 3, the text of B will have to be sent from 3 to 1, re-extending the reference tree of A to include processor 1.



Solid lines denote links in the reference tree of object A, dashed lines the tree for object B. A solid box represents a processor with a text for A, a shaded box a processor with a text for B. Successive reference tree contractions, alternating between the reference trees of A and B, can lead to the sequence of situations shown above as (a) through (f), whereupon a final contraction involving B will restore situation (a).

Figure 3.10: Cyclic restart in garbage collection

Of course, there are many other sequences of events, even starting from one of the configurations shown in Figure 3.10, which will result in both reference trees

converging on the same point; however, it is possible to have an infinitely long sequence of events which never results in either object being collected.

This problem is similar to the problem of *cyclic restart* in some transaction-based data base management systems[28]; perhaps there are solutions to this garbage-collection problem which are analogous to solutions to the cyclic restart problem. For a practical system, it should be quite safe to rely on random timing differences due to changing loads to prevent any continuing pattern such as shown in Figure 3.10 from persisting for long.

3.3.4: Management of Mutable Objects

Immutable objects are the most trouble-free objects to deal with in a distributed system; consequently, much of the foregoing discussion has implicitly been biased toward the management of immutable objects and toward taking advantage of the extra flexibility these objects allow. This section aims to restore the balance by highlighting the ways in which the strategies that have been presented are relevant to mutable objects, and to comment on some interesting alternatives in the application of these strategies to mutable objects.

Mutable objects are objects such as cell, token, and semaphore bodies whose texts may change over time. The obvious and simplest way to manage one of these objects is to keep only one copy of its text, rather than allowing multiple copies. Using this strategy, a processor sending a text for one of these objects would not retain a copy for itself, as it ordinarily might.

3.3.4.1: Management of Tokens

A strategy for managing tokens which is more sophisticated than simply keeping all information about a token in one place has already been suggested. Pieces of the text may be kept in separate places, so long as no piece is ever kept in more than one place. Thus the complete text of the token may not necessarily exist at any one location, but exists as the *union* of all the pieces of text for that token scattered about the system.

The basic approach is to represent the text of a token as a pair of tables as shown in Figure 2.7. Actually, only one table need be kept, with two kinds of entries. Every time the read side of a token receives an object, an RTOK entry naming the object is added to the table for that token. Whenever the write side receives an object, a WTOK entry naming the object is added to the table. This table is the text of an object which we shall call the token *body*. In fact, as described in the previous paragraph, the RTOK and WTOK entries go not into one centralized token table but (at least initially) into the portion of the table present on the processor where the object was received.

The RTOK and WTOK entries satisfy the record-keeping requirements for tokens, but it is difficult to use these entries by themselves to generate the events that should be generated when both the read and write sides of a token have received messages, perhaps on different processors. Without creating duplicate RTOK or WTOK entries on other processors that also have pieces of the text of a token body, those processors must be notified whenever a new RTOK or WTOK entry is made, so that the proper events can be generated. For this purpose, two other kinds of table entries are required. Whenever, for example, an RTOK entry is added to a token body, an entry of type WREQ (request writers) naming the same object is added also. Similarly, addition of a WTOK entry prompts the addition of an

entry of type RREQ. Unlike RTOK and WTOK entries, WREQ and RREQ entries are designed to spread to all parts of the text of a token body. Thus once one of these entries has been added to the token body text on one processor, that processor will send it to all of its neighbors who have pieces of text for that token body, they will send it to all their neighbors who qualify, and so on. Every time a WREQ entry is added to a token table, an event is generated for each WTOK entry that was already present; every time an RREQ entry is added, an event is generated to correspond to each RTOK entry already present.

Since they never interact with any entries added to a table after them, WREQ and RREQ entries can actually be temporary. The only reason they need to stay around even temporarily is to give the processor a chance to forward them to all its neighbors. This, unfortunately, cannot always be done immediately. Since WREQ and RREQ entries are in reality pieces of token body text, they, like any other text, may only be sent over a link when the sender is master of the link. Thus these entries may have to be stored temporarily, and should include a field indicating which neighbors they have not yet been sent to. Once a WREQ or RREQ entry has been sent to all neighbors having part of the text for the token body, however, it can be deleted.

Figure 3.12 illustrates a possible sequence of events for a token known on three processors P1, P2, and P3, which form a chain in which P2 lies between P1 and P3 (shown in Figure 3.11). The horizontal lines separate snapshots of the state of the various token tables through time. The column labeled "Event" shows any events generated from the token at that time. The scenario is that the read side of the token receives object *X* on processor P2 at time T1, and the write side receives object *Y* on processor P1 at time T6.

In more detail, the receipt of object *X* on processor P2 by the read side of the

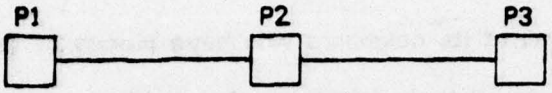


Figure 3.11: Processor configuration for token examples

Time	*Processor* P1	P2	P3	*Event*
T1		RTOK *X* WREQ *X*		
T2		RTOK *X* WREQ *X*	WREQ *X*	
T3		RTOK *X* WREQ *X*		
T4	WREQ *X*	RTOK *X*		
T5		RTOK *X*		
T6	WTOK *Y* RREQ *Y*	RTOK *X*		
T7	WTOK *Y*	RTOK *X* RREQ *Y*		*XY*
T8	WTOK *Y*	RTOK *X*	RREQ *Y*	
T9	WTOK *Y*	RTOK *X*		

Figure 3.12: Example of token operation

token causes the "RTOK *X*" and "WREQ *X*" entries to be added to the token table there, according to the algorithm that has been outlined. At time T2, the WREQ entry has been sent to P3, but must still be kept at P2 since it has not yet been possible to send it to P1. At time T3, the WREQ entry has been deleted at P3 since it has no neighbors to send it to (there is no need to send it to P2 because that is where it came from!). At T4, the WREQ entry has finally been sent to P1

and thus may be deleted at P2. Skipping to T6, we see that the write side of the token has received the object *Y* at P1. By T7 the RREQ entry generated by this occurrence has moved to P2, where it interacts with the previously added RTOK entry to cause the event *XY*. After T7, the RREQ entry moves on to P3 and finally out of the system, leaving only the "RTOK *X*" and "WTOK *Y*" entries as permanent records of this activity. These will come into action in the future if additional objects are received by either the read or write side of the token.

As it has been described up to now, the implementation has a bug, depicted in Figure 3.13.

	Processor			
Time	P1	P2	P3	*Event*
T1	RTOK *X* WREQ *X*		WTOK *Y* RREQ *Y*	
T2	RTOK *X*	WREQ *X* RREQ *Y*	WTOK *Y*	
T3	RTOK *X* RREQ *Y*	WREQ *X*	WTOK *Y*	*XY*
T4	RTOK *X*	WREQ *X*	WTOK *Y*	
T5	RTOK *X*		WTOK *Y* WREQ *X*	*XY*
T6	RTOK *X*		WTOK *Y*	

Figure 3.13: Incorrect token operation

This situation can result if a token receives an object at its read side on one processor and an object at its write side on another processor simultaneously, or at least close enough in time that the RREQ or WREQ messages generated by the first event have not yet had time to propagate to all processors having pieces of text for that token. For example, in Figure 3.13, *X* is received at processor P1 by the read side at the same time as *Y* is received at P3 by the write side. The resultant

machinations cause the event *XY* to be generated twice, when in fact it should be generated only once. Devising a solution to this problem requires taking a closer look at exactly what functions the various parts of our implementation are supposed to serve.

The RTOK and WTOK entries are intended to fulfill the record-keeping requirements of our system. They correspond to the table entries in Figure 2.7, summarizing the history of the token so that it will be known what events to generate in the future. WREQ and RREQ are messengers, signs of activity in the token table—specifically, indicators that a new RTOK or WTOK entry has been added. The sole purpose of the WREQ and RREQ entries is to interact with all *previously added* WTOK or RTOK entries to generate the appropriate events. A WREQ or RREQ entry, however, should *not* interact with a *subsequently* added WTOK or RTOK, because the RREQ or WREQ resulting from the addition of the latter *will* interact with the RTOK or WTOK corresponding to the former. Thus we see that the success of our scheme hinges on the specification of a strict ordering on all RTOK and WTOK entries so that it can be determined which were added prior to any given RREQ or WREQ entry and which may have been added subsequently.

The solution adopted in this case was to *time-stamp* all token table entries using a scheme suggested by Lamport[18]. It is not necessary to use an external source of time for this—any scheme for time-stamping a token will be satisfactory if it yields time stamps which show no inconsistency (failure to increase monotonically) observable in a context where the messages relating to the token body are the only messages traveling between processors. Our scheme involves keeping at the head of each token table on each processor a current value of its time stamp—note that, even for the same token, the time stamp values need not be the same on all processors at all times. Every time a new RTOK or WTOK entry is

added to a table, that table's time stamp is first incremented and the new value becomes the time stamp both of the RTOK or WTOK entry and of the corresponding WREQ or RREQ entry. Every time a table entry is sent, the time stamp of the entry is sent with it, and the time stamp of the table it was sent from is also sent. Every time a table entry is received, the time stamp of the *table* on the receiving processor is set to the maximum of its old time stamp and the table time stamp from the message. The time stamp of the new *entry* in the table is just the time stamp of the entry in the message. One unfortunate problem with time stamps is their propensity to overflow after a while. As with eventcounts[24] a viable, though not elegant, solution is just to allocate enough bits for storage of time stamps that a system could operate for a long time with no overflow.

Now a WREQ or RREQ entry being added to a table will only interact with WTOK or RTOK entries whose time stamps are earlier than that of the WREQ or RREQ entry. Ties are prevented by using the unique ID of the processor where an entry was added as the least significant bits of the entry's time stamp.

Figure 3.14 is a replay of the scenario in Figure 3.13 showing how the addition of time stamps solves the problem demonstrated there. The time stamp of a table is shown in square brackets to the left of the table; the time stamp of each entry is shown following it in the form "(*timestamp,processor ID*)." Although some intermediate steps have been shown, the labels T1 through T6 have been chosen to label the situations corresponding to similarly labeled situations in Figure 3.13. Other than the machinations of time stamps at work, the only difference between the two scenarios is that no event is generated at T5 when time stamps are used. This is because the time stamp of (2,P3) on the WTOK entry is later than the time stamp of (2,P1) on the WREQ entry at processor P3 (this assumes a collating sequence in which P3 > P1—the reverse assumption is also tenable and would

Time	Processor P1	P2	P3	Event
	[1]	[1]	[1]	
	[2] RTOK *X*(2,P1) WREQ *X*(2,P1)	[1]	[1]	
T1	[2] RTOK *X*(2,P1) WREQ *X*(2,P1)	[1]	[2] WTOK *Y*(2,P3) RREQ *Y*(2,P3)	
	[2] RTOK *X*(2,P1)	[2] WREQ *X*(2,P1)	[2] WTOK *Y*(2,P3) RREQ *Y*(2,P3)	
T2	[2] RTOK *X*(2,P1)	[2] WREQ *X*(2,P1) RREQ *Y*(2,P3)	[2] WTOK *Y*(2,P3)	
T3	[2] RTOK *X*(2,P1) RREQ *Y*(2,P3)	[2] WREQ *X*(2,P1)	[2] WTOK *Y*(2,P3)	*XY*
T4	[2] RTOK *X*(2,P1)	[2] WREQ *X*(2,P1)	[2] WTOK *Y*(2,P3)	
T5	[2] RTOK *X*(2,P1)	[2]	[2] WTOK *Y*(2,P3) WREQ *X*(2,P1)	
T6	[2] RTOK *X*(2,P1)	[2]	[2] WTOK *Y*(2,P3)	

Figure 3.14: Corrected token operation

result in an event at T5 but none at T3). Due to space limitations, this example does not show any very exotic instance of time stamps at work; the reader is invited to concoct his own scenarios and see how time stamps would handle them.

This completes our discussion of the minimum required to make tokens work, but there are still improvements that will make the use of tokens more feasible. It has probably already occurred to the reader that the table analogy has some liabilities as a basis for an implementation. There are cases where it is neither necessary nor desirable for a token to remember every object ever sent to it. For example, when a token is used to implement recursion, as in the Y_μ operator of Chapter 2, the write side of the token will receive a message once and never be used again, while the read side will receive another message every time the recursive function is called. If tokens are implemented using tables, the tables will grow and

grow as long as the token is in use. This is clearly undesirable, since alternative strategies for recursion manage to get along without tying up unbounded amounts of storage in this way. It is also unnecessary: the history of objects sent to the read side will never be used because they could only come into play if a message was received by the write side, but this will never happen. Thus the only record that must be kept in this case is of the object that was sent to the write side, which will interact with each new object sent to the read side.

The fact that the read table need no longer be kept, and that any current RTOK entries can actually be deleted, can even be discovered by the system in most cases. In the example of the Y_μ operator, the reason we are sure that the write side of the token will never receive another message is that it becomes inaccessible shortly after receiving its first message. If this is true, the object representing the write side of the token (recall that this is distinct from the read side and also from the token body) will eventually be garbage-collected. When the write side is deleted on the last processor having any references to it, the garbage collector (which can easily detect this event) can send a broadcast message notifying all processors having texts for the token body. This can be done using the same mechanism by which RREQ and WREQ entries are "broadcast" to all such processors. Receipt of this notification will cause the deletion of all RTOK entries and a state change preventing new RTOK entries from being added in the future. This deletion is always safe, provided that the garbage collector notification is not allowed to "pass through" RREQ and WREQ entries awaiting transmission. Of course, all the mechanism described in this paragraph applies symmetrically to the case where the last reference to a read side is deleted and the write side is still active.

Although this implementation of tokens has not been proven correct, it has

been coded and exercised in a situation which seems, from other experience, to have been able to produce almost any pathological sequence of events imaginable. Thus there is reason to be confident that any bugs in the scheme described in this section are the result of flaws in the author's descriptive talents rather than signs of any underlying weakness in the scheme.

3.3.4.2: Management of Cells

Along with tokens, cells can be managed more imaginatively than just by allowing a maximum of one copy of the text. Consider, for example, a cell which contained a number indicating the current year. Such a cell could be thought of as an immutable object for long periods of time, with occasional lapses of mutability around New Year's Day. Similar comments could be made regarding much of the data stored in conventional file systems, such as commands and subroutines provided by the system staff (at least once they seem to work!).

It would be pleasing to treat these as immutable objects during the periods when they are not changing, yet retain the capability to change them if appropriate. Happily, our system is fairly well set up to do just this. It is certainly simple to treat a cell body as if it were immutable, allowing any number of copies of it to be made. The problems start when the cell must be updated. Fortunately, the reference tree for the cell body links together all processors having any direct knowledge of the cell, so all processors in possession of a copy of the cell body's text can be located. Consequently, a processor wishing to perform an update on a cell can broadcast a message to all other processors having copies of the old cell text, inducing them to delete their copies. Meanwhile, of course, the updating processor should not give out any new copies. When acknowledgment is received that all other copies of the cell body text are gone, the update may be performed

on what is at that moment the only copy of the cell body text in the system.

Complications arise, however, if two processors more or less simultaneously decide to perform different updates to the same cell. Presumably all the other processors will delete their copies of the cell body text when they are requested to, but each of the two updating processors will hold fast, waiting for the other to delete its copy of the text—a deadlock. This problem, like the problem of too many events being generated from tokens, can be solved by placing any strict ordering on the update requests. One possibility would be simply to use the processor ID of the processor making the request, but this would probably result in some processors operating at permanent and unfair advantages or disadvantages. A more palatable solution is to have time stamps on cell bodies like the time stamps on token bodies and order updates by these time stamps. Each update message broadcast to all processors to induce them to delete their cell body texts will carry the time stamp of the update. If an update message is received by a processor which is itself attempting an update, the time stamp of the message is compared to that of this processor's update. If less, this processor's update attempt is aborted, to be tried again later. If greater, the message is ignored. This rule insures that the update with the oldest time stamp will have precedence.

The beauty of this scheme for handling cells is that, except for the additional overhead of maintaining a time stamp, it includes as a special case the obvious way of managing cells by only keeping a single copy of the text. It is trivial to tell (by consulting the processor's reference tree status bits for the cell body) whether any other processors have copies of a cell body text. If not, the update can be performed locally just as in the single-text algorithm. Only if other copies exist must other processors be informed. In either case, immediately after an update is performed there is again only one copy of the text, which will only

spread to other processors if it is still in demand.

3.3.4.3: Summary

As this section has been seeking to demonstrate, it is possible by the exercise of some amount of ingenuity to come up with alternative schemes for managing mutable objects which preserve more of the distributed flavor of our system. The bookkeeping associated with the reference tree mechanism was seen to be a material help in this process, facilitating the job of tracking down all other processors with knowledge of a mutable object when a change must be made to that object.

Unfortunately, this section also bears witness to the sensitivity of the design of these alternative schemes to differences in the attributes of the mutable objects being implemented. Not only does the implementation of tokens differ considerably from that of cells, a section on how to implement semaphores would contain yet another *ad hoc* scheme (no such section has been included because it would yield no new insight into the capabilities of the reference tree scheme).

On the bright side, the implementations described for tokens and cells seem quite viable. Both reduce to their simplest cases when only one processor knows about the object (which is likely to be the case for the majority of objects), seeming not to carry an excessive penalty for their generality when that generality is not used. For the most part, the implementation of tokens avoids retaining data in token tables that will never be used, and thus makes it reasonable to actually use tokens for all the purposes discussed in Chapter 2. The suggested implementation of cells may make it much less painful to retain an option to change an object, even if that option is not expected to be exercised often. It also raises other possibilities, for example, objects analogous to files on current timesharing systems

(objects of type, say, mutable character string). In these objects, it would be possible to reach inside and change part of the text without altering the rest. If this semantics were deemed useful, there is no reason why such objects could not be implemented in much the same manner as cells.

3.3.5: Alternative Reference Tree Algorithms

The purpose of this section is to deal with a couple of questions that were postponed from the middle of the foregoing presentation of reference trees. The first topic we discuss is removing the requirement that all reference trees remain connected.

3.3.5.1: Disconnecting Reference Trees

An objection to the current reference tree mechanism is that if two distant processors need to know about an object, a whole chain of intermediate processors are forced to know about it also, due to the requirement that reference trees must remain connected. Much of the overhead imposed on these processors might be eliminated if this connection could be broken. It is not difficult to devise a protocol by which one of the intermediate processors could cause this to happen. Since all communication involving an object travels only along links in its reference tree, however, two requirements must be met: a custodian of a copy of the text of the object must exist on either side of the break (otherwise the processors in one of the disconnected pieces would have no access to the text of the object), and the object must be immutable (otherwise an update performed in one half of the tree would never become visible in the other half). Effectively, breaking a link in the reference tree for an object creates two reference trees for the object. Each of the new trees will then behave as if it were the only reference tree for

that object. Specifically, leaf nodes of either tree may then delete themselves from it, so all the intermediate processors in our example can leave the tree, one by one, resulting in the desired situation where only the two distant processors know about the object.

In fact, the only problem with disconnecting a reference tree arises if it is ever desired to re-connect the tree. The reference tree management protocol avoids cycles in reference trees by refusing to make a connection if two branches of reference tree for the same object bump into each other. This is done because it is assumed that all branches of reference tree for the same object are already connected; therefore, adding another connection would close a cycle. If, as the result of a disconnection, the two branches are not connected, the protocol will still refuse to connect them. It would be quite difficult to allow such branches to be re-connected without introducing the possibility that cycles could be formed. Thus it seems that once a reference tree is broken into two or more pieces, those pieces must continue to exist independently for as long as they continue to exist. This is not necessarily bad, however. Each piece is still free to grow, shrink, and move just as the original was, and thus each may independently be reclaimed by the garbage collection mechanism when its usefulness is ended.

3.3.5.2: Reorganizing Reference Trees

Figure 3.4 gave an example of a non-optimal reference tree, and the accompanying text suggested that there might be ways of improving such trees. Such strategies were not investigated in the course of this research, but a couple of approaches to the problem have suggested themselves. Any approach based on purely local knowledge of the reference tree should fit easily into our scheme, provided it preserves the essential properties of reference trees (connectedness and

freedom from cycles).

One possibility is for a leaf node which is aware that one of its neighbors is connected to the tree by a different path (perhaps because of receiving an R+ message from that neighbor) to break its old connection and connect instead to that neighbor. This kind of operation is depicted in Figure 3.15.

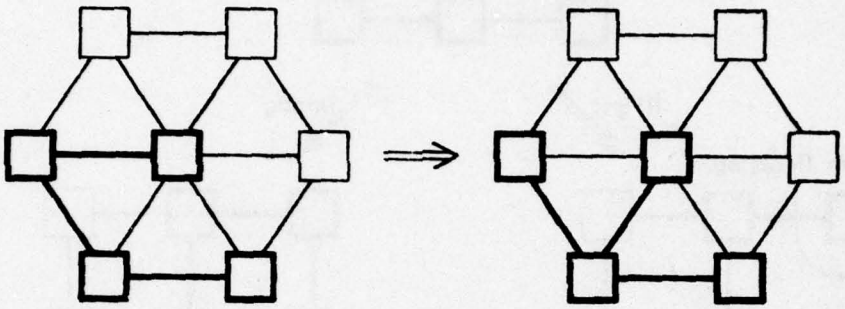


Figure 3.15: A simple reference tree reorganization

It is difficult, unfortunately, for a non-leaf node to make this kind of jump, because it will not know which of its old links to break. If the wrong choice is made, not only will the reference tree become disconnected, but one half of it will contain a cycle, as shown in Figure 3.16.

In addition to the mechanics of reorganizing the tree, there is of course a strategy question—when is this wise? Once again, in simple cases the answer can be fairly obvious, but in general it may not be. If the goal for the processor changing its links is to get closer to a copy of the text of the object, then it is obviously a good idea to change if the processor being connected to has a copy of the text. If it does not have a copy of the text, then it will either have to have some idea how far the nearest text is (a piece of information that might become obsolete every time a text moved) or other considerations will have to be invoked.

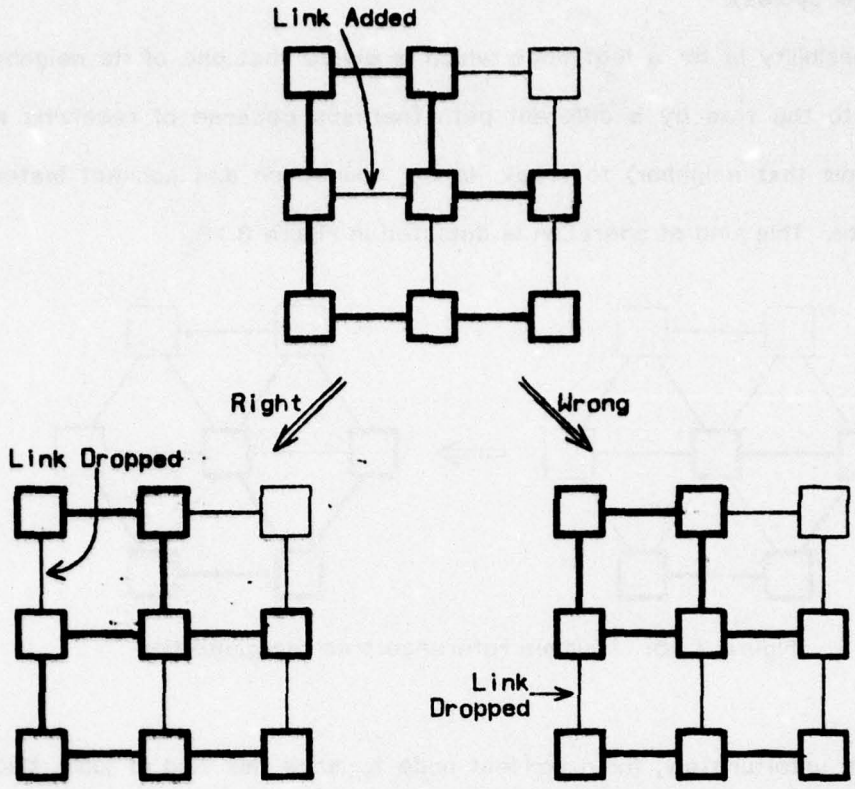


Figure 3.16: A dangerous reference tree reorganization

3.3.6: Summary

This section of the thesis dealt in depth with the question of object management in our distributed system. Reference trees were introduced as a means for keeping track of objects and the processors that know about them. A detailed section described a set of protocols for managing these reference trees in a distributed fashion; both management of membership in a tree and management of object texts were discussed. Special attention was given to the problem of managing texts of mutable objects, holding out hope that such objects can be effectively supported on our distributed system. Reference trees lead naturally to a certain approach to garbage collection on distributed systems, which was

described. Finally, the feasibility of improving reference trees by allowing various reorganizations was briefly explored.

Any problem whose solution is as intricate as that presented here obviously will yield to many other solutions as well. The purpose of this section was to give a concrete lower bound on the effectiveness with which all the problems surrounding object management can be solved. By describing an approach that has actually been implemented, it invites others to duplicate or improve on it. At the same time, it should be recognized that object management is not the only problem facing a distributed system. The object management algorithms presented can assure that the system will operate correctly, but they cannot assure that it will operate efficiently.

3.4: Event Distribution Strategy

It should be clear by now that the algorithm for assigning events to processors is a critical factor in determining the success or failure of the system. In fact, it is undoubtedly the single most important influence on the performance of the system. The object management algorithms described in this chapter are significant in that they assure the correct operation of the system. No matter how good they are, however, the system will not perform satisfactorily unless events are distributed intelligently. Along with being a critical component of the system, unfortunately, event distribution strategy is also a difficult topic to approach rigorously. Thus it seems likely that the measure of an event distribution strategy will only be found by experiment. This section describes the extremely simple strategy that was used in the course of this research, and gives what limited conclusions can be drawn as to its usefulness. Also presented are, several ideas (as yet untested) which may help in the effort to construct more effective event distribution

strategies.

3.4.1: The QFUDGE Strategy

The simple strategy which was actually used was just this: if the number of events on a processor's event list exceeds the number of events on some neighbor's event list by a certain constant (dubbed QFUDGE) or more, then the processor will send a new event to that neighbor rather than appending it to its own event list. In order to keep current about the number of events on each others' event lists, processors include that information about themselves in all messages sent to other processors. If a processor has not sent any messages over a particular link for a while, it will send a null message in that direction just to keep the processor at the other end informed of the state of its business.

The principal motivation behind adopting the QFUDGE strategy was that it was simple and yet produced passable performance, and thus could be used while the other aspects of the implementation were worked out and debugged. To give a rough idea of the effectiveness of this strategy, we present some simulation results. The simulation was of a ring of eight processors, with two different choices for the bandwidth of the interprocessor links. One used relatively high-bandwidth channels, on the order perhaps of one-megabaud links between microprocessors. The other used channels fifty times slower, or approximately twenty kilobaud compared to typical microprocessor computing speeds. The parameter that was actually varied was a ratio between computing speed and channel speed, so the estimates given above must be considered extremely coarse and highly dependent on the computing speed of the interpreters implemented on the processors.

The system was given twenty independent computations of the value of a function (always applied to the same argument) and the simulated elapsed time

was measured from the point where these twenty tasks were given to the system (all at the same processor) to the point where the last of the twenty answers was typed. The test was then repeated with twenty more parallel computations of the same function value. The purpose of this repetition was to gauge the effect of the fact that, after the first test, several object texts useful for the second test (such as the text of the function) would have become available on several of the processors, decreasing the time required for an event being moved to acquire a working set on its new processor. The combined times for the original test and repetition also give some indication of the probable performance of the system had the original work load been twice as severe.

Table 3.17 gives separately the times for the first and second tests, using both fast and slow communication channels, for QFUDGE values of one through five. The times given have been normalized so that unity represents the time required to do all calculations on a single processor.

		QFUDGE Value					*Optimum*
Channel	*Test*	1	2	3	4	5	*QFUDGE*
slow	first	1.65	1.77	1.55	1.22	0.99	>5
slow	second	0.78	0.61	1.51	0.51	0.70	3-4
fast	first	0.68	0.54	0.64	0.53	0.61	3-4
fast	second	0.32	0.31	0.41	0.46	0.53	2

Table 3.17: Execution time using the QFUDGE strategy

The most egregious anomaly in these figures is the entry 1.51 for the execution time of the second test using the slow channel with a QFUDGE value of 3. This particular entry corresponds to a peculiar run in which nineteen of the twenty answers were typed out in a normalized execution time of 0.47. Computation of the remaining result apparently required amazing amounts of interprocessor communication to complete. Using a figure closer to 0.47 for that value makes the table

look nicer, but it is still dangerous to try drawing too many inferences from the information. Some intuitively appealing relationships do seem to be apparent, however. The execution times in each row of Table 3.17 except the first seem to exhibit a minimum, indicating an optimal value of QFUDGE for that application. (We may infer that the first row of figures also has a minimum, to the right of the table at some QFUDGE value greater than 5.) These optimal values are shown to the right of the table.

We may regard too low a value of QFUDGE as encouraging excessive communication overhead, and too high a value as allowing insufficient access to the computing power in other parts of the system. It is thus pleasing (though not surprising) that faster communications seem to favor lower values of QFUDGE, and that the second trials, where less overhead is required to set up working sets, do also.

The figures in Table 3.17 also illustrate both the potential of multiprocessor systems such as the one simulated, and the amount of room that is left for improvement. Under appropriate circumstances, it was indeed possible to cut total elapsed computation time by a factor of three. On the other hand, eight processors were employed to accomplish this (thus an optimal arrangement of events might be expected to cut computation time by a factor of eight). It must also be noted that the kind of trials conducted were of the most favorable possible sort to the system. Much work on scheduling remains to be done before a system such as this can operate effectively on a wide range of problems.

3.4.2: Improved Event Distribution Strategies

We can imagine work on improving the performance of a mu-calculus system being directed along two somewhat interrelated dimensions: improvements to the interpreter algorithm and improvements to the interpreter strategy. The former includes such possibilities as sending several related texts in response to a single inquiry and has already been alluded to. The purpose of this section is to concentrate on the latter.

The magnitude of the job facing an event distribution strategy can be appreciated by thinking of a system computing a weather forecast, or perhaps a solution to Laplace's equation. Such computations are characteristically organized as a grid of logical "cells," each with a certain amount of state information (e.g., temperature, pressure, or wind direction), and each interacting only with its neighbors following a simple set of rules (e.g., difference equations derived from the laws of physics). Stepping such a grid through its paces causes the state variables to converge on the desired solution (e.g., tomorrow's weather forecast). A program of this sort could be written in the mu calculus by creating an object to model each grid point, and arranging the behavior of these objects so that at each iteration each object gave rise to an event sending to each of its neighbor objects the value of each of its relevant state variables. This kind of problem seems like an ideal one for our network to solve, *provided* that events and objects are allocated to processors in a reasonable manner. A reasonable manner would probably be a distribution which related the logical structure of the problem to the physical structure of the system, for example, mapping several grid points onto each processor in such a way that neighboring grid points were assigned either to the same processor or to neighboring processors.

It is fairly likely that the simple QFUDGE strategy would be a miserable failure

in this respect. Assuming that all events and objects started out on one processor, they would be randomly farmed out to its neighbors, and their neighbors, and so on, with almost no regard to their logical relationships. As a result, communication paths would probably become many times longer than necessary. Perhaps the most practical solution to this particular problem would be to specify a language in which the programmer could indicate the optimum distribution. We do not pursue this approach for two reasons: it requires the programmer to know about the physical structure of the system, an attribute we have been trying to hide from him, and it may not help very much in most cases, where the logical structure of the problem is probably more fluid and less known. Another possibility might be some kind of centralized scheduling processor with an overview of all system operations. We do not pursue this either, since it is at variance with the basic system philosophy that all responsibility should be distributed.

It would be impressive to come up with an automatic, distributed scheduling algorithm that caused the event and object distribution for, say, a weather forecasting program, to converge toward the optimum. A more modest objective, however, would just be a scheduling strategy that avoided doing obviously stupid things, like keeping two related objects far apart in the system while messages were constantly being sent back and forth. To this end, we might consider some kind of "rubber band" strategy, wherein objects referenced in an event, especially the receiver object of an event, could exert a certain amount of "pull" on the event, making it easier to ship it to a neighbor where the object was known, and harder to send it where the object was not known. The pull might be especially strong it that event was the only reason for the object's being known on the current processor.

Another possibility might be for the event distribution strategy to take some

account of the difficulty of accumulating a working set. Perhaps an event descended from a long line which had been proceeding smoothly, with no pauses to fetch data from another processor, would be kept at the same location in preference to a newly received event, or an event whose working set appeared not to be present.

Finally, event distribution strategy should be concerned with optimizing more than just CPU usage. Memory usage is another important parameter. The distribution of objects in the system is governed to a considerable extent by the distribution of events. Thus an event distribution strategy might try to gauge the quantity of objects required to be near each event (this is probably the same as the working set of the event) and try to collect an appropriate mix on each processor, depending on each processor's memory size and computing speed.

3.4.3: Storage Management Strategy

The job of the garbage collector is the identification and removal of objects that are no longer accessible in the system. Storage management strategy goes beyond this to include also the handling of objects which, although still accessible, are not likely to be needed at a particular processor. A useful view of the memory of any individual processor in the system is as a cache containing, ideally, the data upon which the processor is most probably about to operate. If a processor requires access to information (i.e., an object text) not available in its "cache," then that data must be fetched from elsewhere in the system.

In light of the similarity between storage management strategy and garbage collection, it is reasonable to assign to the garbage collector the chief responsibility for managing this cache, and in particular for pruning dead wood from it to make room for new growth. The discussion in this section will assume a traditional kind

of mark-and-sweep garbage collector[17] which runs at intervals to free up memory space that is not being used productively. Most of the considerations mentioned, however, also apply to a real-time incremental garbage collector such as that described by Baker[1].

The first step for the garbage collector running on a processor is to determine which objects are being used on that processor and which are not. This may be done by starting with the event list and marking every object referenced from any event. If the text of any of these objects is present, all objects referenced in that text should likewise be marked, and so on. This marking phase is not sufficient in itself, however. Any objects having texts on this processor but also known on other processors must also be marked, along with all objects reachable from them. This is because, even though none of the objects so marked may be reachable from any event on this processor, the globally-known object may be reachable from an event active on some other processor, whereupon objects reachable from this object will also be accessible to that event, even though these latter objects might currently be known only on this processor. It may be desirable to employ two different kinds of marks in this stage, since the marks will be used for more than one purpose (cache management as well as outright deletion of objects).

Any object not marked at all during this procedure must be known only on this processor (since all objects known elsewhere were marked) and must not be accessible from any event on this processor or any object known on any other processor. Such an object is therefore garbage and can be deleted. This is not the only way the garbage collector can reclaim storage, however. If there are any objects known also on other processors and not directly reachable from any event on this processor, it may be appropriate to reduce this processor's involvement

with those objects. If this processor has copies of any of their texts, it is probably desirable to delete those copies or send them elsewhere if possible. If this processor is a leaf node in any of their reference trees, it is most likely worthwhile for this processor to remove itself from those reference trees altogether.

There may be objects on this processor which are not known on any other processor and are not directly reachable from any event on this processor, and yet are not garbage-collectable. These objects must be kept because they were marked in the second phase described above as being reachable from other objects known outside this processor. Since they cannot be deleted outright, part of the storage management strategy question for these objects is whether to keep them at their present location or send them to a neighbor. This question cannot be answered on the basis of statistics gathered by the garbage collector (at least not of the form described above) and should presumably be answered after considering the amount of storage available on this processor and its neighbors, or on the basis of some estimate of which processor is most likely to eventually use these objects. In the simple approximation used in the simulation of our system, such objects were allowed to stay put. Note that if any other processor actually asks for these objects, they will then be known on more than one processor and, at the next garbage collection, may be shipped away from the processor that had been assuming a storage role.

Another category of object the garbage collector may want to deal with is composed of objects that are reachable from active events on the processor but are distantly enough related to those events that it may not be worthwhile to keep them close at hand. Our simulation simply left these alone, but a more sophisticated algorithm might take into account some of the considerations mentioned

above.

Even with the simple strategies actually used, typical garbage collections reclaimed half or more of the space in use at the time the garbage collection started—probably an indication that a relatively large proportion of objects have short lifetimes and never become known outside the processor where they were created. Of course, this statistic is probably also an indication that the amount of available space on each processor vastly exceeded the amount needed for the computations that were attempted. At any rate, the inclusion of good storage management heuristics in the garbage collector can probably assist a good event distribution strategy to achieve optimum results.

3.4.4: Conclusions Regarding Event Distribution

We have discussed the importance of good event distribution and storage management strategies and some approaches to their design. The performance of the simple QFUDGE event distribution strategy was evaluated. This strategy falls far short of obtaining maximum system throughput, but does manage to make the system run considerably faster than a single processor. An analysis of some of the demands facing event distribution strategies led to the conclusion that some kind of "rubber band" strategy might be an improvement. Also discussed were the utility of using working set and memory use statistics. Hopefully, improved strategies using these ideas can be combined with improvements in algorithms for accumulating working sets to give a much more convincing demonstration of the validity of the kind of multiprocessor system described in this chapter.

3.5: Conclusions Regarding Our Implementation

This chapter has been devoted to outlining a possible implementation of the mu calculus on a distributed system and commenting on plausible alternative implementations. Although the description has not been at the bit-diddling level, all significant aspects of the implementation have been discussed; a simulator for a multiple-processor implementation using these algorithms has actually been constructed.

The implementation described in this chapter is a system based on object references that uses an event list to keep track of pending computations. The physical structure assumed is a cellular network in which each processor has a certain limited number of immediate neighbors with which it can communicate directly. A system standard external representation for object references, object texts, and processor ID's is assumed, although their internal representations can differ. Object management is done by means of reference trees, which can be maintained by means of a strictly distributed algorithm (no centralized control) using protocols presented in the chapter. Garbage collection and management of mutable objects can also be integrated naturally into the reference tree scheme. Finally, a simple event distribution strategy was described and more effective ones suggested.

The simple strategies used in the simulated implementation were sufficient to yield a significant speed increase in some cases over a single processor following the same algorithm. It is much more difficult, of course, to arrive at meaningful figures comparing the performance of this system to that of entirely differently organized systems solving the same problems. Since the approach presented here is very general, it is probably safe to say that, for any particular problem, it is possible to find an implementation more efficient than that given in this thesis, just as

special-purpose hardware can usually be constructed to solve a problem more quickly than a general-purpose computer. However, general-purpose computers have the advantage of flexibility, and thus there is a market for them. With additional refinement, there is reason to hope that multiprocessor systems of the kind described in this chapter can be realistic entrants into the general-purpose multiprocessor system sweepstakes, although they will still undoubtedly be outperformed in specific applications by more specialized systems.

Chapter 4: Conclusions and Directions for Future Work

This thesis can be taken as a study of two fairly unrelated concepts: the semantics of message passing, and the architecture of an object-reference system for multiple processors. Both of these areas of research, however, are dictated by our top-level goal of developing a methodology for replacing single large computers with networks of smaller computers.

The mu calculus set forth in Chapter 2 supplies a semantic basis in terms of which programs to be run on such a system can be expressed. A very spare language, the mu calculus is not intended for direct use as a programming tool, but rather as a target language for translators which might be developed. Thus the primary emphasis is not on style but on the variety of semantic elements included. These elements form a useful set, allowing realistic programs to be written while also providing some novel capabilities (tokens) useful on distributed systems. The mu calculus is also of interest in its own right as a simple model of message-passing computation.

Chapter 3 describes a distributed object reference system which could be used to implement the mu calculus. The system architecture chosen is a cellular network of processors, with each processor having a limited number of nearest neighbors. Although many other architectures are possible, most can be made to imitate this architecture. Furthermore, a large enough network of processors (e.g., exceeding the capacity of an individual Ethernet or ring) is almost forced to have this kind of architecture, at least if viewed at the proper level of aggregation. The implementation described includes algorithms for the assignment of objects and events to processors and for doing the necessary bookkeeping. Simulations of the implementation show that significant use can be made of the parallel capabilities of

a network based on this implementation, but that the implementation is still far from using the network to its full potential.

In the introduction to this thesis, much is made of the assertion that what the world of distributed computing needs is a system designed from the top down, rather than from the bottom up. That top-down approach may have been obscured somewhat in the pages that follow, thus it is appropriate to review it. Our top-level goal is the desire to be able to use many parallel processors effectively; the mu calculus is in the middle of the hierarchy, and our implementation is at the bottom.

How then do we go from this top-level goal to the realization described in this thesis? The original stimulus for this research was the problem of replacing one large processor with a multitude of smaller processors having roughly similar total capacity. One motivation for this is that the collection of smaller, slower processors is likely to cost less; another is the ease of adding or removing increments of computing power in such a system. Obviously, such a network can only be successful if enough of the smaller processors can be kept busy. In other words, a sufficient amount of parallelism must be found in the tasks given to the system that its tremendous capacity can actually be utilized. There are various ways in which parallelism might be discovered in programs written for a single sequential machine, or programs might be written with more parallelism explicit. In the opinion of the author, the second alternative is the more promising, but the thesis does not take a position on this issue. Rather, the first step in the top-down development is the specification of a language in which this parallelism can be expressed well, whatever the route by which it may have been arrived at. This language is the mu calculus.

Having defined the mu calculus as an embodiment of the semantic concepts that should be supported by our network, the next step in the top-down

development is to design the network. The mechanics of the mu calculus favor an object-reference system with garbage-collected storage. Additionally, we should like the allocation of objects and events to processors to be as flexible as possible, to give the maximum latitude for rearranging things to keep all processors as busy as possible. These desiderata, in turn, are all satisfied by the reference tree mechanism for object management.

4.1: Alternatives to Our Design

Our design is certainly not the only possible result of a top-down effort to design a distributed system. Obviously, if the objective of the design effort were different, for example, the development of a special-purpose rather than a general-purpose system, the result could differ. Even given the same objective, there are many ways to satisfy it. The full generality of the reference tree mechanism may be unwarranted in many cases, where a simple strategy of permanently or semi-permanently attaching an object to a designated processor might seem preferable. Garbage collection may often be considered an unnecessary luxury, and manual storage management adopted instead. Changes such as these can be worked into the reference tree scheme without requiring a complete redesign. The use of object references, at least at some level of aggregation of data, can be a great aid in communicating between processors, and probably would be retained in almost *any design.*

Even the use of message passing as a semantic basis for the system is more a matter of style than content. The mu calculus seems to do the best at incorporating those features which raise interesting points about the design of distributed systems and leaving out those which only result in uninteresting detail. It is thus well suited to an exposition of these points, such as this thesis. The

advantage of the mu calculus is in allowing us a very fine-grained look at a computation--there is no mechanism hidden in, say, subroutine calls and returns, or expression evaluation. We can therefore avoid explicit discussion of how these might be implemented, and concentrate on more primitive mechanisms. This is not to say that expression evaluations and subroutine calls and returns are not occurring, only that they are happening by means of series of simpler operations whose implementations have already been specified. Thus the mu calculus is a good tool for exposing the organizational aspects relevant to the system design, although not necessarily the best foundation for a particular system.

On the other hand, the mu calculus is well matched to the kinds of activities that are important in distributed systems. The breaking down of, for example, expression evaluations and subroutine entry and exit into more elementary operations increases the number of options available to a distributed system in handling these activities.

Other than the choice of a semantic basis, the major decision point at which many alternatives are available is in the selection of a network topology to form the hardware base for the distributed system. This choice is affected by economics and scale, as well as by the range of applications envisioned. For a small network, shared memory might be the most appropriate, offering high bandwidth and essentially zero communication delay. For a network of modest size, a ring or Ethernet might be a good choice. However, there are technological limitations on the number of processors that can be attached to one ring or Ethernet. There is also the fact that the ring or Ether will tend to become a bottleneck as more processors are attached to it. Thus the cellular network assumed in this thesis seems to be the only topology with the potential of scaling up indefinitely. Of course, it might be more effective for nodes in the network to be, say, small clusters of

processors with shared memory, and negotiations internal to such a cluster might be conducted on a different basis. Nevertheless, our network philosophy seems to support the greatest range of scaling up and down, both in size and performance. It has the additional advantage of conforming to the "thin-wire" philosophy articulated by Metcalfe[21]; thus strategies suitable for it are likely to be applicable to, if not optimal for, almost any network.

Once the basic design decision to support mu-calculus-like message passing on a cellular network has been made, the number of possible ways to fill out the picture dwindles, although there is doubtless still plenty of scope for ingenuity. This thesis has shown one way to proceed from this design decision to filling in the rest of the picture. The implementation presented is complete; it has been tested and found able to make at least some use of a network of processors. Its potential, however, is much greater. Further research promises to yield implementations that come much closer to realizing this potential.

4.2: The Mu Calculus

There are several possible avenues for the further development of the mu calculus. The basic definition of the pure mu calculus plus tokens seems fairly sound, but undoubtedly has many interesting properties that have not yet been discovered. Evaluations of the mu calculus as a tool for understanding message passing might be especially welcome. The translations between the mu and lambda calculi were only informally justified. Formal correctness proofs of these would give additional insight into the relationship of the two calculi. Proofs of various properties of the actors constructed out of tokens, such as the parallelism actor π and the mu-calculus Y operator, would help in understanding these actors.

A better axiom scheme for handling cells and other mutable objects would be a

great contribution. The scheme given in this thesis is adequate for describing formally what these objects do, but is of little use in proving anything interesting. Proofs of properties of, for example, the arbiter actor α, would also be illuminating. Work done by Hewitt[16] and Greif[11] is relevant here.

Finally, the development of the mu calculus into a humanly usable programming language is a necessary prerequisite to actually building and using any system based on the mu calculus.

4.3: Implementations

The possibilities for further work in the design of distributed systems are almost too numerous to mention. Other object management schemes are possible; perhaps a more compact object management protocol can be developed. Great amounts of creativity can be absorbed in the development of new strategies for managing various kinds of mutable objects. Schemes for reorganizing reference trees may substantially improve the performance of the system. In fact, even the gathering of statistics to gauge the possible effect of such strategies would be a significant step. The garbage collection scheme presented in this thesis is at best imperfectly understood. Are there real cases in which some garbage will never be collected? Also, how can it be modified to incorporate an incremental garbage collector such as that of Baker[1]?

The development of improved event distribution and storage management strategies promises to have a major effect on system performance. This is probably the area of research in which additional results are the most crucially needed.

Finally, all these suggestions pertain only to making improvements on the design presented in this thesis. The scope for imagination and originality in devising new approaches to distributed computing is practically limitless.

Appendix A: Correctness of the Membership Protocol

This appendix tells the story of how the reference tree membership protocol was developed and tested. Originally, a much simpler membership protocol (having only four states instead of thirteen) was invented and used as the basis for an early implementation of a simulator for a multiple-processor message-passing system. After a protracted period of debugging failed to produce reliable operation, the author began to suspect the protocol itself. This led to the construction of a LISP program for testing protocols. The ability of this program to find weaknesses in the protocol and enable tracing of the circumstances under which these weaknesses would manifest themselves was invaluable in arriving at the protocol finally used. This appendix describes the structure and use of this protocol-testing program, both for its own interest and for the confidence it gives us that the protocols presented in this thesis are correct.

A.1: The Protocol-Testing Program

The reference tree protocols deal with the state of an individual node in a network and how it reacts to various stimuli; the protocol-testing program uses the state-transition rules in a protocol to enumerate the possible states for an entire network. The state of a network is considered to be an element of the cross product of the possible states of all nodes in the network and the possible instantaneous contents of all communication links in the network (i.e., all messages sent but not yet received). There is no reason to assume that the set of possible states of a network is finite—it may be possible for unbounded numbers of messages to pile up on the links. In fact, the original reference tree membership protocol led to networks with this property. Of course, this makes it impossible for the protocol

tester to enumerate all possible states.

Fortunately, there are protocols which confine networks using them to have only finite numbers of possible states (i.e., states accessible from some approved starting state). These protocols can be analyzed exhaustively and thus are more attractive in the absence of good theoretical tools for studying protocols. Even though the number of accessible network states may be finite, it may still be quite large, especially if the network is complex. Furthermore, it is not obvious how a test using one particular network topology generalizes to other topologies. For these and other reasons, the protocol tester was actually given an abstraction of a network in which attention was focused on just two adjacent nodes and the link between them. Assumptions about the behavior of the network were introduced by restricting the rules for spontaneous state transitions, leaving only those transitions that might actually be caused by the hypothesized activities of the rest of the network. The choice of these assumptions was governed by properties being tested (e.g., consistency, resistance to disconnecting the tree, etc.).

Thus the "network" for our protocol tester consists of two nodes (which we may call A and B) and the link between them (which is actually modeled as a pair of FIFO message queues, one containing messages traveling from A to B, the other containing messages traveling from B to A—let us call these queues AB and BA, respectively). The complete state of this network is a combination of the individual states of nodes A and B, and the contents of queues AB and BA, whose elements are significant both by their identity and their position (the assumption is that messages never "pass" each other—no message is ever received before another message sent earlier in the same direction). A state transition for the network occurs any time a state transition rule from the protocol (as restricted by assumptions about the rest of the network) is applied to either A or B. If a rule calls for a

spontaneous transition of, say, A, then the new network state is derived from the old network state by changing the state of A to its new value and appending the output (if any) accompanying the transition to A's output queue AB. If the transition is caused by receiving a message, then that message must be at the head of the processor's input queue (BA in the case of A), whereupon that message will be removed from the queue, the state transition effected, and the resulting output (if any) appended to the output queue as before. The network state after the transition is the state resulting after all three of these operations have been performed.

The protocol tester simply applies all applicable rules to each state in a set of initial states to generate a larger set of accessible states. The process is then repeated for each new state added to this set until applying any applicable rule to any state in the set always yields another state already in the set. At this point the procedure terminates and the final set may be inspected to see if it contains any undesirable members or does not contain some members that were expected.

Obviously the output from this program is only as valid as the assumptions used in formulating the state transition rules, so the results necessarily constitute at best an informal demonstration of the soundness of the protocols unless the assumptions are rigorously justified. Acceptable results of tests with this program have combined, however, with the positive empirical evidence gained by using the protocols to give the author reasonable confidence that the protocol is correct; several bugs were found in the process of implementing the protocol finally chosen, and in every instance the problem was found to be a failure to implement correctly the desired protocol, rather than a weakness in the protocol itself.

A.2: Testing the Membership Protocol

A test of the membership protocol should look for several properties:

* closure—all states reachable from any possible starting state should be states that can be handled following the rules of the protocol; no processor should ever receive a message that it cannot handle in its current state.

* consistency—all stable network states (i.e., states in which all message queues are empty) reachable from any possible starting state should show the desired relationship between processor states; for example, we would not like to see a stable state in which two processors each thought they were masters of the same link.

* resistance to disconnection—we do not want our protocol to allow a reference tree to become disconnected.

* resistance to forming cycles—we do not want our protocol to allow undirected cycles to be formed in a reference tree.

The test described in this section was primarily concerned with demonstrating closure and consistency—the most obvious trouble spots in earlier protocols. Comments will be made later about resistance to disconnection and forming cycles.

For this test, state X? of the membership protocol was split into two states X? and XI. The new state XI follows state transition rules similar to X?, but records the fact that a processor in state X? received a reference to the object after entering that state. Although it does not formally change the state of the processor with respect to that object, receiving a reference while in state X? modifies the behavior of the processor in several ways. For one, since the processor once

again has a reference to the object, it can now send one back, whereas a processor in state X? that has not received a reference should have no references to that object to send (having just finished indicating its desire to leave the reference tree for that object altogether). The second difference is that a processor in state XI should not ever receive an A- message, as a processor in state X? might. This is because an A- message in this context finalizes a deletion from the reference tree, which is improper if a processor in state X? has once again come into possession of an object reference. Once again, it is important to note that the addition of "state" XI is simply an accounting maneuver to give us a clearer perception of what is happening to the system—it does not imply that a processor in operation explicitly needs to keep track of whether it is in state X? or XI. The mechanism that keeps reference trees connected is present in other states (notably SR and SR?); the purpose of introducing XI is only to check (for our test) that this is really happening correctly.

We now give the state transition rules used in the membership protocol test; additional comments regarding the relevance of this particular set will appear with the justifications of individual rules. Rules input to the protocol tester have the following form:

(*current-state input output next-state*)

An *input* entry of nil indicates a spontaneous transition; an *output* entry of nil indicates a "quiet" transition—one that is not accompanied by any output.

The following rules were taken directly from the definition of the membership protocol (Table 3.6):

(X nil nil N)	(S - A- N?1)	(N?1 nil R+ M?2)	(M?2 nil R- M?2)
(X R+ + S)	(S R- nil S)	(N?1 A- nil N)	(M?2 A- nil M?)
(N nil nil X)	(SR nil R- SR)	(N?1 R- nil N?1)	(M?2 R- nil M?2)
(N nil R+ M?)	(SR + nil M)	(M? nil R- M?)	(S? nil R- SR?)
(N R+ - N?1)	(SR - A+ S?)	(M? R+ - M?1)	(S? A+ nil S)
(M nil R- M)	(SR R- nil SR)	(M? + nil M)	(S? A- A- N)
(M nil + S)	(N? nil nil X?)	(M? - A- N)	(S? R- nil S?)
(M nil - X?)	(N? nil R- N?)	(M? R- nil M?)	(SR? nil R- SR?)
(M R- nil M)	(N? A+ A- N?1)	(M?1 nil R- M?1)	(SR? A+ nil SR)
(S nil R- SR)	(N? A- A- N)	(M?1 - A- N?1)	(SR? A- A- N)
(S + nil M)	(N? R- nil N?)	(M?1 R- nil M?1)	(SR? R- nil SR?)

For state X? we have the following rules:

(X? nil nil N?) (X? A+ A+ M) (X? A- A- X) (X? R- nil X!)

These rules are the same as the state transition rules given in Table 3.6 for state X?, except that the transition upon receiving an R- message is to the parallel state X!, recording the receipt of an object reference, instead of back to X?. Also, the transition "(X? nil R- X?)" has been omitted, since it can only occur if the processor has received a reference to the object after its transition to state X?. In our accounting, such a processor would be in state X! instead.

The rules for state X! are

(X! nil nil N?) (X! nil R- X!) (X! A+ A+ M) (X! R- nil X!)

These rules are derived from the rules for state X? by changing all references to X? into X!—once an object reference has been received while in state X?, that fact will be remembered until a further transition (to N? or M) occurs. The transition "(X! A- A- X)" has been omitted, since an A- message (finalizing a break in the tree) should never be received if an object reference was received first.

This completes our presentation of the state transition rules used for testing the membership protocol; however, a few more explanations should be made before the output of that program is shown. There is one exception to the rule

that the protocol tester records all messages in each *message queue* individually in the order in which they were sent. Since sending any number of consecutive R- messages has the same effect on the sender's state as sending a single one, any sequence of consecutive R- messages in a queue is represented as a single R- message (it is actually this compression which makes the number of accessible network states finite). As a result, transitions triggered by the receipt of an R- message can happen in two ways: accompanied by the removal of the R- message (the proper action if it was a single message) or without disturbing the R- message (the effect of only removing the first from a stream of several). The protocol tester must use both methods to ensure that all accessible states are found.

The second shortcut used by the protocol tester was based on the observation that the two-node "network" under consideration is symmetrical. Since the set of starting states exhibits the same symmetry, this symmetry could be used to reduce by half the number of network states actually considered. Although this technique was used to speed things up during the protocol design, the tabulation below will be presented without taking advantage of this compression, to make it more usable as a reference.

The representation used for a network state was

(A-*state* AB-*queue* B-*state* BA-*queue*)

Within a queue, the rightmost element was the first sent (and will be the first received). Thus the network state

(M? (R- R+) X ())

means that processor A has sent an R+ message followed by some number of R- messages and is now in state M?. Processor B has not yet received any of these

messages, has not sent any messages that have not been received by A, and is currently in state X.

The single initial network state used was

(X () X ())

which led to all the others shown below. The network states are shown sorted, first by the state of processor A, then by the state of processor B, then by the contents of the message queues. Each line in the following tabulation contains a reference number, a possible network state, the reference number of a previous state that could lead to that state, the transition rule that would be used to go from that previous state to the current state, and the processor to which the rule would be applied.

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
1.	(M () S ())	51	(M? + nil M)	A
2.	(M (R-) S ())	52	(M? + nil M)	A
3.	(M (R- A+) S? ())	4	(M nil R- M)	A
4.	(M (A+) S? ())	599	(X! A+ A+ M)	A
5.	(M (R- A+ R-) S? ())	6	(M nil R- M)	A
6.	(M (A+ R-) S? ())	601	(X! A+ A+ M)	A
7.	(M () SR ())	8	(M R- nil M)	A
8.	(M () SR (R-))	1	(S nil R- SR)	B
9.	(M (R-) SR ())	10	(M R- nil M)	A
10.	(M (R-) SR (R-))	2	(S nil R- SR)	B
11.	(M (R- A+) SR? ())	12	(M R- nil M)	A
12.	(M (R- A+) SR? (R-))	3	(S? nil R- SR?)	B
13.	(M (A+) SR? ())	14	(M R- nil M)	A
14.	(M (A+) SR? (R-))	4	(S? nil R- SR?)	B
15.	(M (R- A+ R-) SR? ())	16	(M R- nil M)	A
16.	(M (R- A+ R-) SR? (R-))	5	(S? nil R- SR?)	B
17.	(M (A+ R-) SR? ())	18	(M R- nil M)	A
18.	(M (A+ R-) SR? (R-))	6	(S? nil R- SR?)	B
19.	(M? (R- R+) M? (R- R+))	20	(M? nil R- M?)	B
20.	(M? (R- R+) M? (R+))	43	(N nil R+ M?)	B
21.	(M? (R+) M? (R- R+))	22	(M? nil R- M?)	B
22.	(M? (R+) M? (R+))	44	(N nil R+ M?)	B
23.	(M? () M?1 (R- - R+))	24	(M?1 nil R- M?1)	B
24.	(M? () M?1 (- R+))	22	(M? R+ - M?1)	B
25.	(M? () M?1 (R- - R- R+))	26	(M?1 nil R- M?1)	B
26.	(M? () M?1 (- R- R+))	21	(M? R+ - M?1)	B
27.	(M? (R-) M?1 (R- - R+))	28	(M?1 nil R- M?1)	B
28.	(M? (R-) M?1 (- R+))	20	(M? R+ - M?1)	B
29.	(M? (R-) M?1 (R- - R- R+))	30	(M?1 nil R- M?1)	B
30.	(M? (R-) M?1 (- R- R+))	19	(M? R+ - M?1)	B
31.	(M? () M?2 (R- R+ -))	32	(M?2 nil R- M?2)	B
32.	(M? () M?2 (R+ -))	45	(N?1 nil R+ M?2)	B
33.	(M? (R- R+ A-) M?2 (R- R+))	34	(M?2 nil R- M?2)	B
34.	(M? (R- R+ A-) M?2 (R+))	46	(N?1 nil R+ M?2)	B
35.	(M? (R+ A-) M?2 (R- R+))	36	(M?2 nil R- M?2)	B
36.	(M? (R+ A-) M?2 (R+))	47	(N?1 nil R+ M?2)	B
37.	(M? (R- R+ A- R-) M?2 (R- R+))	38	(M?2 nil R- M?2)	B
38.	(M? (R- R+ A- R-) M?2 (R+))	48	(N?1 nil R+ M?2)	B
39.	(M? (R+ A- R-) M?2 (R- R+))	40	(M?2 nil R- M?2)	B
40.	(M? (R+ A- R-) M?2 (R+))	49	(N?1 nil R+ M?2)	B
41.	(M? (R-) M?2 (R- R+ -))	42	(M?2 nil R- M?2)	B
42.	(M? (R-) M?2 (R+ -))	50	(N?1 nil R+ M?2)	B
43.	(M? (R- R+) N ())	55	(X nil nil N)	B
44.	(M? (R+) N ())	56	(X nil nil N)	B
45.	(M? () N?1 (-))	44	(N R+ - N?1)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
46.	(M? (R- R+ A-) N?1 ())	47	(M? nil R- M?)	A
47.	(M? (R+ A-) N?1 ())	252	(N nil R+ M?)	A
48.	(M? (R- R+ A- R-) N?1 ())	49	(M? nil R- M?)	A
49.	(M? (R+ A- R-) N?1 ())	253	(N nil R+ M?)	A
50.	(M? (R-) N?1 (-))	43	(N R+ - N?1)	B
51.	(M? () S (+))	56	(X R+ + S)	B
52.	(M? (R-) S (+))	55	(X R+ + S)	B
53.	(M? () SR (R- +))	51	(S nil R- SR)	B
54.	(M? (R-) SR (R- +))	52	(S nil R- SR)	B
55.	(M? (R- R+) X ())	56	(M? nil R- M?)	A
56.	(M? (R+) X ())	254	(N nil R+ M?)	A
57.	(M?1 (R- - R+) M? ())	59	(M?1 nil R- M?1)	A
58.	(M?1 (R- - R+) M? (R-))	57	(M? nil R- M?)	B
59.	(M?1 (- R+) M? ())	22	(M? R+ - M?1)	A
60.	(M?1 (- R+) M? (R-))	59	(M? nil R- M?)	B
61.	(M?1 (R- - R- R+) M? ())	63	(M?1 nil R- M?1)	A
62.	(M?1 (R- - R- R+) M? (R-))	61	(M? nil R- M?)	B
63.	(M?1 (- R- R+) M? ())	20	(M? R+ - M?1)	A
64.	(M?1 (- R- R+) M? (R-))	63	(M? nil R- M?)	B
65.	(M?1 (R- -) M?1 (R- -))	66	(M?1 nil R- M?1)	B
66.	(M?1 (R- -) M?1 (-))	57	(M? R+ - M?1)	B
67.	(M?1 (R- -) M?1 (R- - R-))	68	(M?1 nil R- M?1)	B
68.	(M?1 (R- -) M?1 (- R-))	58	(M? R+ - M?1)	B
69.	(M?1 (-) M?1 (R- -))	70	(M?1 nil R- M?1)	B
70.	(M?1 (-) M?1 (-))	59	(M? R+ - M?1)	B
71.	(M?1 (-) M?1 (R- - R-))	72	(M?1 nil R- M?1)	B
72.	(M?1 (-) M?1 (- R-))	60	(M? R+ - M?1)	B
73.	(M?1 (R- - R-) M?1 (R- -))	74	(M?1 nil R- M?1)	B
74.	(M?1 (R- - R-) M?1 (-))	61	(M? R+ - M?1)	B
75.	(M?1 (R- - R-) M?1 (R- - R-))	76	(M?1 nil R- M?1)	B
76.	(M?1 (R- - R-) M?1 (- R-))	62	(M? R+ - M?1)	B
77.	(M?1 (- R-) M?1 (R- -))	78	(M?1 nil R- M?1)	B
78.	(M?1 (- R-) M?1 (-))	63	(M? R+ - M?1)	B
79.	(M?1 (- R-) M?1 (R- - R-))	80	(M?1 nil R- M?1)	B
80.	(M?1 (- R-) M?1 (- R-))	64	(M? R+ - M?1)	B
81.	(M?1 () M?2 (R- R+ A- -))	82	(M?2 nil R- M?2)	B
82.	(M?1 () M?2 (R+ A- -))	113	(N?1 nil R+ M?2)	B
83.	(M?1 () M?2 (R- R+ A- R- -))	84	(M?2 nil R- M?2)	B
84.	(M?1 () M?2 (R+ A- R- -))	114	(N?1 nil R+ M?2)	B
85.	(M?1 () M?2 (R- R+ A- - R-))	86	(M?2 nil R- M?2)	B
86.	(M?1 () M?2 (R+ A- - R-))	115	(N?1 nil R+ M?2)	B
87.	(M?1 () M?2 (R- R+ A- R- - R-))	88	(M?2 nil R- M?2)	B
88.	(M?1 () M?2 (R+ A- R- - R-))	116	(N?1 nil R+ M?2)	B
89.	(M?1 (R- - R+ A-) M?2 ())	91	(M?1 nil R- M?1)	A
90.	(M?1 (R- - R+ A-) M?2 (R-))	89	(M?2 nil R- M?2)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
91.	(M?1 (- R+ A-) M?2 ())	36	(M? R+ - M?1)	A
92.	(M?1 (- R+ A-) M?2 (R-))	91	(M?2 nil R- M?2)	B
93.	(M?1 (R- - R- R+ A-) M?2 ())	95	(M?1 nil R- M?1)	A
94.	(M?1 (R- - R- R+ A-) M?2 (R-))	93	(M?2 nil R- M?2)	B
95.	(M?1 (- R- R+ A-) M?2 ())	34	(M? R+ - M?1)	A
96.	(M?1 (- R- R+ A-) M?2 (R-))	95	(M?2 nil R- M?2)	B
97.	(M?1 (R- - R+ A- R-) M?2 ())	99	(M?1 nil R- M?1)	A
98.	(M?1 (R- - R+ A- R-) M?2 (R-))	97	(M?2 nil R- M?2)	B
99.	(M?1 (- R+ A- R-) M?2 ())	40	(M? R+ - M?1)	A
100.	(M?1 (- R+ A- R-) M?2 (R-))	99	(M?2 nil R- M?2)	B
101.	(M?1 (R- - R- R+ A- R-) M?2 ())	103	(M?1 nil R- M?1)	A
102.	(M?1 (R- - R- R+ A- R-) M?2 (R-))	101	(M?2 nil R- M?2)	B
103.	(M?1 (- R- R+ A- R-) M?2 ())	38	(M? R+ - M?1)	A
104.	(M?1 (- R- R+ A- R-) M?2 (R-))	103	(M?2 nil R- M?2)	B
105.	(M?1 (R-) M?2 (R- R+ A- -))	106	(M?2 nil R- M?2)	B
106.	(M?1 (R-) M?2 (R+ A- -))	117	(N?1 nil R+ M?2)	B
107.	(M?1 (R-) M?2 (R- R+ A- R- -))	108	(M?2 nil R- M?2)	B
108.	(M?1 (R-) M?2 (R+ A- R- -))	118	(N?1 nil R+ M?2)	B
109.	(M?1 (R-) M?2 (R- R+ A- - R-))	110	(M?2 nil R- M?2)	B
110.	(M?1 (R-) M?2 (R+ A- - R-))	119	(N?1 nil R+ M?2)	B
111.	(M?1 (R-) M?2 (R- R+ A- R- - R-))	112	(M?2 nil R- M?2)	B
112.	(M?1 (R-) M?2 (R+ A- R- - R-))	120	(N?1 nil R+ M?2)	B
113.	(M?1 () N?1 (A- -))	70	(M?1 - A- N?1)	B
114.	(M?1 () N?1 (A- R- -))	69	(M?1 - A- N?1)	B
115.	(M?1 () N?1 (A- - R-))	72	(M?1 - A- N?1)	B
116.	(M?1 () N?1 (A- R- - R-))	71	(M?1 - A- N?1)	B
117.	(M?1 (R-) N?1 (A- -))	66	(M?1 - A- N?1)	B
118.	(M?1 (R-) N?1 (A- R- -))	65	(M?1 - A- N?1)	B
119.	(M?1 (R-) N?1 (A- - R-))	68	(M?1 - A- N?1)	B
120.	(M?1 (R-) N?1 (A- R- - R-))	67	(M?1 - A- N?1)	B
121.	(M?2 (R- R+ -) M? ())	123	(M?2 nil R- M?2)	A
122.	(M?2 (R- R+ -) M? (R-))	121	(M? nil R- M?)	B
123.	(M?2 (R+ -) M? ())	309	(N?1 nil R+ M?2)	A
124.	(M?2 (R+ -) M? (R-))	123	(M? nil R- M?)	B
125.	(M?2 (R- R+) M? (R- R+ A-))	126	(M? nil R- M?)	B
126.	(M?2 (R- R+) M? (R+ A-))	197	(N nil R+ M?)	B
127.	(M?2 (R- R+) M? (R- R+ A- R-))	128	(M? nil R- M?)	B
128.	(M?2 (R- R+) M? (R+ A- R-))	198	(N nil R+ M?)	B
129.	(M?2 (R+) M? (R- R+ A-))	130	(M? nil R- M?)	B
130.	(M?2 (R+) M? (R+ A-))	199	(N nil R+ M?)	B
131.	(M?2 (R+) M? (R- R+ A- R-))	132	(M? nil R- M?)	B
132.	(M?2 (R+) M? (R+ A- R-))	200	(N nil R+ M?)	B
133.	(M?2 () M?1 (R- - R+ A-))	134	(M?1 nil R- M?1)	B
134.	(M?2 () M?1 (- R+ A-))	130	(M? R+ - M?1)	B
135.	(M?2 () M?1 (R- - R- R+ A-))	136	(M?1 nil R- M?1)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
136.	(M?2 () M?1 (- R- R+ A-))	129	(M? R+ - M?1)	B
137.	(M?2 () M?1 (R- - R+ A- R-))	138	(M?1 nil R- M?1)	B
138.	(M?2 () M?1 (- R+ A- R-))	132	(M? R+ - M?1)	B
139.	(M?2 () M?1 (R- - R- R+ A- R-))	140	(M?1 nil R- M?1)	B
140.	(M?2 () M?1 (- R- R+ A- R-))	131	(M? R+ - M?1)	B
141.	(M?2 (R- R+ A- -) M?1 ())	143	(M?2 nil R- M?2)	A
142.	(M?2 (R- R+ A- -) M?1 (R-))	141	(M?1 nil R- M?1)	B
143.	(M?2 (R+ A- -) M?1 ())	311	(N?1 nil R+ M?2)	A
144.	(M?2 (R+ A- -) M?1 (R-))	143	(M?1 nil R- M?1)	B
145.	(M?2 (R- R+ A- R- -) M?1 ())	147	(M?2 nil R- M?2)	A
146.	(M?2 (R- R+ A- R- -) M?1 (R-))	145	(M?1 nil R- M?1)	B
147.	(M?2 (R+ A- R- -) M?1 ())	313	(N?1 nil R+ M?2)	A
148.	(M?2 (R+ A- R- -) M?1 (R-))	147	(M?1 nil R- M?1)	B
149.	(M?2 (R- R+ A- - R-) M?1 ())	151	(M?2 nil R- M?2)	A
150.	(M?2 (R- R+ A- - R-) M?1 (R-))	149	(M?1 nil R- M?1)	B
151.	(M?2 (R+ A- - R-) M?1 ())	315	(N?1 nil R+ M?2)	A
152.	(M?2 (R+ A- - R-) M?1 (R-))	151	(M?1 nil R- M?1)	B
153.	(M?2 (R- R+ A- R- - R-) M?1 ())	155	(M?2 nil R- M?2)	A
154.	(M?2 (R- R+ A- R- - R-) M?1 (R-))	153	(M?1 nil R- M?1)	B
155.	(M?2 (R+ A- R- - R-) M?1 ())	317	(N?1 nil R+ M?2)	A
156.	(M?2 (R+ A- R- - R-) M?1 (R-))	155	(M?1 nil R- M?1)	B
157.	(M?2 (R-) M?1 (R- - R+ A-))	158	(M?1 nil R- M?1)	B
158.	(M?2 (R-) M?1 (- R+ A-))	126	(M? R+ - M?1)	B
159.	(M?2 (R-) M?1 (R- - R- R+ A-))	160	(M?1 nil R- M?1)	B
160.	(M?2 (R-) M?1 (- R- R+ A-))	125	(M? R+ - M?1)	B
161.	(M?2 (R-) M?1 (R- - R+ A- R-))	162	(M?1 nil R- M?1)	B
162.	(M?2 (R-) M?1 (- R+ A- R-))	128	(M? R+ - M?1)	B
163.	(M?2 (R-) M?1 (R- - R- R+ A- R-))	164	(M?1 nil R- M?1)	B
164.	(M?2 (R-) M?1 (- R- R+ A- R-))	127	(M? R+ - M?1)	B
165.	(M?2 () M?2 (R- R+ - A-))	166	(M?2 nil R- M?2)	B
166.	(M?2 () M?2 (R+ - A-))	205	(N?1 nil R+ M?2)	B
167.	(M?2 () M?2 (R- R+ - A- R-))	168	(M?2 nil R- M?2)	B
168.	(M?2 () M?2 (R+ - A- R-))	206	(N?1 nil R+ M?2)	B
169.	(M?2 (R- R+ - A-) M?2 ())	171	(M?2 nil R- M?2)	A
170.	(M?2 (R- R+ - A-) M?2 (R-))	169	(M?2 nil R- M?2)	B
171.	(M?2 (R+ - A-) M?2 ())	319	(N?1 nil R+ M?2)	A
172.	(M?2 (R+ - A-) M?2 (R-))	171	(M?2 nil R- M?2)	B
173.	(M?2 (R- R+ A-) M?2 (R- R+ A-))	174	(M?2 nil R- M?2)	B
174.	(M?2 (R- R+ A-) M?2 (R+ A-))	207	(N?1 nil R+ M?2)	B
175.	(M?2 (R- R+ A-) M?2 (R- R+ A- R-))	176	(M?2 nil R- M?2)	B
176.	(M?2 (R- R+ A-) M?2 (R+ A- R-))	208	(N?1 nil R+ M?2)	B
177.	(M?2 (R+ A-) M?2 (R- R+ A-))	178	(M?2 nil R- M?2)	B
178.	(M?2 (R+ A-) M?2 (R+ A-))	209	(N?1 nil R+ M?2)	B
179.	(M?2 (R+ A-) M?2 (R- R+ A- R-))	180	(M?2 nil R- M?2)	B
180.	(M?2 (R+ A-) M?2 (R+ A- R-))	210	(N?1 nil R+ M?2)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
181.	(M?2 (R- R+ - A- R-) M?2 ())	183	(M?2 nil R- M?2)	A
182.	(M?2 (R- R+ - A- R-) M?2 (R-))	181	(M?2 nil R- M?2)	B
183.	(M?2 (R+ - A- R-) M?2 ())	325	(N?1 nil R+ M?2)	A
184.	(M?2 (R+ - A- R-) M?2 (R-))	183	(M?2 nil R- M?2)	B
185.	(M?2 (R- R+ A- R-) M?2 (R- R+ A-))	186	(M?2 nil R- M?2)	B
186.	(M?2 (R- R+ A- R-) M?2 (R+ A-))	211	(N?1 nil R+ M?2)	B
187.	(M?2 (R- R+ A- R-) M?2 (R- R+ A- R-))	188	(M?2 nil R- M?2)	B
188.	(M?2 (R- R+ A- R-) M?2 (R+ A- R-))	212	(N?1 nil R+ M?2)	B
189.	(M?2 (R+ A- R-) M?2 (R- R+ A-))	190	(M?2 nil R- M?2)	B
190.	(M?2 (R+ A- R-) M?2 (R+ A-))	213	(N?1 nil R+ M?2)	B
191.	(M?2 (R+ A- R-) M?2 (R- R+ A- R-))	192	(M?2 nil R- M?2)	B
192.	(M?2 (R+ A- R-) M?2 (R+ A- R-))	214	(N?1 nil R+ M?2)	B
193.	(M?2 (R-) M?2 (R- R+ - A-))	194	(M?2 nil R- M?2)	B
194.	(M?2 (R-) M?2 (R+ - A-))	215	(N?1 nil R+ M?2)	B
195.	(M?2 (R-) M?2 (R- R+ - A- R-))	196	(M?2 nil R- M?2)	B
196.	(M?2 (R-) M?2 (R+ - A- R-))	216	(N?1 nil R+ M?2)	B
197.	(M?2 (R- R+) N (A-))	121	(M? - A- N)	B
198.	(M?2 (R- R+) N (A- R-))	122	(M? - A- N)	B
199.	(M?2 (R+) N (A-))	123	(M? - A- N)	B
200.	(M?2 (R+) N (A- R-))	124	(M? - A- N)	B
201.	(M?2 (R- R+ A-) N? ())	241	(X? nil nil N?)	B
202.	(M?2 (R- R+ A-) N? (R-))	201	(N? nil R- N?)	B
203.	(M?2 (R+ A-) N? ())	243	(X? nil nil N?)	B
204.	(M?2 (R+ A-) N? (R-))	203	(N? nil R- N?)	B
205.	(M?2 () N?1 (- A-))	199	(N R+ - N?1)	B
206.	(M?2 () N?1 (- A- R-))	200	(N R+ - N?1)	B
207.	(M?2 (R- R+ A-) N?1 (A-))	141	(M?1 - A- N?1)	B
208.	(M?2 (R- R+ A-) N?1 (A- R-))	142	(M?1 - A- N?1)	B
209.	(M?2 (R+ A-) N?1 (A-))	143	(M?1 - A- N?1)	B
210.	(M?2 (R+ A-) N?1 (A- R-))	144	(M?1 - A- N?1)	B
211.	(M?2 (R- R+ A- R-) N?1 (A-))	145	(M?1 - A- N?1)	B
212.	(M?2 (R- R+ A- R-) N?1 (A- R-))	146	(M?1 - A- N?1)	B
213.	(M?2 (R+ A- R-) N?1 (A-))	147	(M?1 - A- N?1)	B
214.	(M?2 (R+ A- R-) N?1 (A- R-))	148	(M?1 - A- N?1)	B
215.	(M?2 (R-) N?1 (- A-))	197	(N R+ - N?1)	B
216.	(M?2 (R-) N?1 (- A- R-))	198	(N R+ - N?1)	B
217.	(M?2 () S (+ A-))	239	(X R+ + S)	B
218.	(M?2 () S (+ A- R-))	240	(X R+ + S)	B
219.	(M?2 (R-) S (+ A-))	237	(X R+ + S)	B
220.	(M?2 (R-) S (+ A- R-))	238	(X R+ + S)	B
221.	(M?2 (R- R+ A-) S? ())	222	(M?2 nil R- M?2)	A
222.	(M?2 (R+ A-) S? ())	339	(N?1 nil R+ M?2)	A
223.	(M?2 (R- R+ A- R-) S? ())	224	(M?2 nil R- M?2)	A
224.	(M?2 (R+ A- R-) S? ())	340	(N?1 nil R+ M?2)	A
225.	(M?2 () SR (R- + A-))	217	(S nil R- SR)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
226.	(M?2 () SR (R- + A- R-))	218	(S nil R- SR)	B
227.	(M?2 (R-) SR (R- + A-))	219	(S nil R- SR)	B
228.	(M?2 (R-) SR (R- + A- R-))	220	(S nil R- SR)	B
229.	(M?2 (R- R+ A-) SR? ())	230	(M?2 R- nil M?2)	A
230.	(M?2 (R- R+ A-) SR? (R-))	221	(S? nil R- SR?)	B
231.	(M?2 (R+ A-) SR? ())	232	(M?2 R- nil M?2)	A
232.	(M?2 (R+ A-) SR? (R-))	222	(S? nil R- SR?)	B
233.	(M?2 (R- R+ A- R-) SR? ())	234	(M?2 R- nil M?2)	A
234.	(M?2 (R- R+ A- R-) SR? (R-))	223	(S? nil R- SR?)	B
235.	(M?2 (R+ A- R-) SR? ())	236	(M?2 R- nil M?2)	A
236.	(M?2 (R+ A- R-) SR? (R-))	224	(S? nil R- SR?)	B
237.	(M?2 (R- R+) X (A-))	197	(N nil nil X)	B
238.	(M?2 (R- R+) X (A- R-))	198	(N nil nil X)	B
239.	(M?2 (R+) X (A-))	199	(N nil nil X)	B
240.	(M?2 (R+) X (A- R-))	200	(N nil nil X)	B
241.	(M?2 (R- R+ A-) X? ())	243	(M?2 nil R- M?2)	A
242.	(M?2 (R- R+ A-) X? (R-))	202	(N? nil nil X?)	B
243.	(M?2 (R+ A-) X? ())	347	(N?1 nil R+ M?2)	A
244.	(M?2 (R+ A-) X? (R-))	204	(N? nil nil X?)	B
245.	(N () M? (R- R+))	246	(M? nil R- M?)	B
246.	(N () M? (R+))	251	(N nil R+ M?)	B
247.	(N (A-) M?2 (R- R+))	248	(M?2 nil R- M?2)	B
248.	(N (A-) M?2 (R+))	252	(N?1 nil R+ M?2)	B
249.	(N (A- R-) M?2 (R- R+))	250	(M?2 nil R- M?2)	B
250.	(N (A- R-) M?2 (R+))	253	(N?1 nil R+ M?2)	B
251.	(N () N ())	254	(X nil nil N)	B
252.	(N (A-) N?1 ())	45	(M? - A- N)	A
253.	(N (A- R-) N?1 ())	50	(M? - A- N)	A
254.	(N () X ())	598	(X nil nil N)	A
255.	(N? () M?2 (R- R+ A-))	256	(M?2 nil R- M?2)	B
256.	(N? () M?2 (R+ A-))	259	(N?1 nil R+ M?2)	B
257.	(N? (R-) M?2 (R- R+ A-))	258	(M?2 nil R- M?2)	B
258.	(N? (R-) M?2 (R+ A-))	260	(N?1 nil R+ M?2)	B
259.	(N? () N?1 (A-))	262	(S - A- N?1)	B
260.	(N? (R-) N?1 (A-))	261	(S - A- N?1)	B
261.	(N? (R- -) S ())	262	(N? nil R- N?)	A
262.	(N? (-) S ())	638	(X? nil nil N?)	A
263.	(N? (R- - R-) S ())	264	(N? nil R- N?)	A
264.	(N? (- R-) S ())	640	(X? nil nil N?)	A
265.	(N? () S? (A+))	279	(SR - A+ S?)	B
266.	(N? () S? (A+ R-))	280	(SR - A+ S?)	B
267.	(N? (R- - A+) S? ())	268	(N? nil R- N?)	A
268.	(N? (- A+) S? ())	644	(X? nil nil N?)	A
269.	(N? (R- - R- A+) S? ())	270	(N? nil R- N?)	A
270.	(N? (- R- A+) S? ())	646	(X? nil nil N?)	A

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
271.	(N? (R- - A+ R-) S? ())	272	(N? nil R- N?)	A
272.	(N? (- A+ R-) S? ())	648	(X? nil nil N?)	A
273.	(N? (R- - R- A+ R-) S? ())	274	(N? nil R- N?)	A
274.	(N? (- R- A+ R-) S? ())	650	(X? nil nil N?)	A
275.	(N? (R-) S? (A+))	277	(SR - A+ S?)	B
276.	(N? (R-) S? (A+ R-))	278	(SR - A+ S?)	B
277.	(N? (R- -) SR ())	278	(N? R- nil N?)	A
278.	(N? (R- -) SR (R-))	261	(S nil R- SR)	B
279.	(N? (-) SR ())	280	(N? R- nil N?)	A
280.	(N? (-) SR (R-))	262	(S nil R- SR)	B
281.	(N? (R- - R-) SR ())	282	(N? R- nil N?)	A
282.	(N? (R- - R-) SR (R-))	263	(S nil R- SR)	B
283.	(N? (- R-) SR ())	284	(N? R- nil N?)	A
284.	(N? (- R-) SR (R-))	264	(S nil R- SR)	B
285.	(N? () SR? (R- A+))	265	(S? nil R- SR?)	B
286.	(N? () SR? (R- A+ R-))	266	(S? nil R- SR?)	B
287.	(N? (R- - A+) SR? ())	288	(N? R- nil N?)	A
288.	(N? (R- - A+) SR? (R-))	267	(S? nil R- SR?)	B
289.	(N? (- A+) SR? ())	290	(N? R- nil N?)	A
290.	(N? (- A+) SR? (R-))	268	(S? nil R- SR?)	B
291.	(N? (R- - R- A+) SR? ())	292	(N? R- nil N?)	A
292.	(N? (R- - R- A+) SR? (R-))	269	(S? nil R- SR?)	B
293.	(N? (- R- A+) SR? ())	294	(N? R- nil N?)	A
294.	(N? (- R- A+) SR? (R-))	270	(S? nil R- SR?)	B
295.	(N? (R- - A+ R-) SR? ())	296	(N? R- nil N?)	A
296.	(N? (R- - A+ R-) SR? (R-))	271	(S? nil R- SR?)	B
297.	(N? (- A+ R-) SR? ())	298	(N? R- nil N?)	A
298.	(N? (- A+ R-) SR? (R-))	272	(S? nil R- SR?)	B
299.	(N? (R- - R- A+ R-) SR? ())	300	(N? R- nil N?)	A
300.	(N? (R- - R- A+ R-) SR? (R-))	273	(S? nil R- SR?)	B
301.	(N? (- R- A+ R-) SR? ())	302	(N? R- nil N?)	A
302.	(N? (- R- A+ R-) SR? (R-))	274	(S? nil R- SR?)	B
303.	(N? (R-) SR? (R- A+))	275	(S? nil R- SR?)	B
304.	(N? (R-) SR? (R- A+ R-))	276	(S? nil R- SR?)	B
305.	(N?1 () M? (R- R+ A-))	306	(M? nil R- M?)	B
306.	(N?1 () M? (R+ A-))	331	(N nil R+ M?)	B
307.	(N?1 () M? (R- R+ A- R-))	308	(M? nil R- M?)	B
308.	(N?1 () M? (R+ A- R-))	332	(N nil R+ M?)	B
309.	(N?1 (-) M? ())	246	(N R+ - N?1)	A
310.	(N?1 (-) M? (R-))	309	(M? nil R- M?)	B
311.	(N?1 (A- -) M?1 ())	70	(M?1 - A- N?1)	A
312.	(N?1 (A- -) M?1 (R-))	311	(M?1 nil R- M?1)	B
313.	(N?1 (A- R- -) M?1 ())	66	(M?1 - A- N?1)	A
314.	(N?1 (A- R- -) M?1 (R-))	313	(M?1 nil R- M?1)	B
315.	(N?1 (A- - R-) M?1 ())	78	(M?1 - A- N?1)	A

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
316.	(N?1 (A- - R-) M?1 (R-))	315	(M?1 nil R- M?1)	B
317.	(N?1 (A- R- - R-) M?1 ())	74	(M?1 - A- N?1)	A
318.	(N?1 (A- R- - R-) M?1 (R-))	317	(M?1 nil R- M?1)	B
319.	(N?1 (- A-) M?2 ())	248	(N R+ - N?1)	A
320.	(N?1 (- A-) M?2 (R-))	319	(M?2 nil R- M?2)	B
321.	(N?1 (A-) M?2 (R- R+ A-))	322	(M?2 nil R- M?2)	B
322.	(N?1 (A-) M?2 (R+ A-))	335	(N?1 nil R+ M?2)	B
323.	(N?1 (A-) M?2 (R- R+ A- R-))	324	(M?2 nil R- M?2)	B
324.	(N?1 (A-) M?2 (R+ A- R-))	336	(N?1 nil R+ M?2)	B
325.	(N?1 (- A- R-) M?2 ())	250	(N R+ - N?1)	A
326.	(N?1 (- A- R-) M?2 (R-))	325	(M?2 nil R- M?2)	B
327.	(N?1 (A- R-) M?2 (R- R+ A-))	328	(M?2 nil R- M?2)	B
328.	(N?1 (A- R-) M?2 (R+ A-))	337	(N?1 nil R+ M?2)	B
329.	(N?1 (A- R-) M?2 (R- R+ A- R-))	330	(M?2 nil R- M?2)	B
330.	(N?1 (A- R-) M?2 (R+ A- R-))	338	(N?1 nil R+ M?2)	B
331.	(N?1 () N (A-))	309	(M? - A- N)	B
332.	(N?1 () N (A- R-))	310	(M? - A- N)	B
333.	(N?1 (A-) N? ())	347	(X? nil nil N?)	B
334.	(N?1 (A-) N? (R-))	333	(N? nil R- N?)	B
335.	(N?1 (A-) N?1 (A-))	311	(M?1 - A- N?1)	B
336.	(N?1 (A-) N?1 (A- R-))	312	(M?1 - A- N?1)	B
337.	(N?1 (A- R-) N?1 (A-))	313	(M?1 - A- N?1)	B
338.	(N?1 (A- R-) N?1 (A- R-))	314	(M?1 - A- N?1)	B
339.	(N?1 (A-) S? ())	265	(N? A+ A- N?1)	A
340.	(N?1 (A- R-) S? ())	275	(N? A+ A- N?1)	A
341.	(N?1 (A-) SR? ())	342	(N?1 R- nil N?1)	A
342.	(N?1 (A-) SR? (R-))	339	(S? nil R- SR?)	B
343.	(N?1 (A- R-) SR? ())	344	(N?1 R- nil N?1)	A
344.	(N?1 (A- R-) SR? (R-))	340	(S? nil R- SR?)	B
345.	(N?1 () X (A-))	331	(N nil nil X)	B
346.	(N?1 () X (A- R-))	332	(N nil nil X)	B
347.	(N?1 (A-) X? ())	384	(S - A- N?1)	A
348.	(N?1 (A-) X? (R-))	334	(N? nil nil X?)	B
349.	(S () M ())	351	(M? + nil M)	B
350.	(S () M (R-))	352	(M? + nil M)	B
351.	(S (+) M? ())	590	(X R+ + S)	A
352.	(S (+) M? (R-))	351	(M? nil R- M?)	B
353.	(S (+ A-) M?2 ())	592	(X R+ + S)	A
354.	(S (+ A-) M?2 (R-))	353	(M?2 nil R- M?2)	B
355.	(S (+ A- R-) M?2 ())	594	(X R+ + S)	A
356.	(S (+ A- R-) M?2 (R-))	355	(M?2 nil R- M?2)	B
357.	(S () N? (R- -))	358	(N? nil R- N?)	B
358.	(S () N? (-))	384	(X? nil nil N?)	B
359.	(S () N? (R- - R-))	360	(N? nil R- N?)	B
360.	(S () N? (- R-))	386	(X? nil nil N?)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
361.	(S () S (+))	349	(M nil + S)	B
362.	(S () S (+ R-))	350	(M nil + S)	B
363.	(S (+) S ())	1	(M nil + S)	A
364.	(S (+ R-) S ())	2	(M nil + S)	A
365.	(S (+ A+) S? ())	4	(M nil + S)	A
366.	(S (+ R- A+) S? ())	3	(M nil + S)	A
367.	(S (+ A+ R-) S? ())	6	(M nil + S)	A
368.	(S (+ R- A+ R-) S? ())	5	(M nil + S)	A
369.	(S () SR (R- +))	361	(S nil R- SR)	B
370.	(S () SR (R- + R-))	362	(S nil R- SR)	B
371.	(S (+) SR ())	372	(S R- nil S)	A
372.	(S (+) SR (R-))	363	(S nil R- SR)	B
373.	(S (+ R-) SR ())	374	(S R- nil S)	A
374.	(S (+ R-) SR (R-))	364	(S nil R- SR)	B
375.	(S (+ A+) SR? ())	376	(S R- nil S)	A
376.	(S (+ A+) SR? (R-))	365	(S? nil R- SR?)	B
377.	(S (+ R- A+) SR? ())	378	(S R- nil S)	A
378.	(S (+ R- A+) SR? (R-))	366	(S? nil R- SR?)	B
379.	(S (+ A+ R-) SR? ())	380	(S R- nil S)	A
380.	(S (+ A+ R-) SR? (R-))	367	(S? nil R- SR?)	B
381.	(S (+ R- A+ R-) SR? ())	382	(S R- nil S)	A
382.	(S (+ R- A+ R-) SR? (R-))	368	(S? nil R- SR?)	B
383.	(S () X? (R- -))	357	(N? nil nil X?)	B
384.	(S () X? (-))	349	(M nil - X?)	B
385.	(S () X? (R- - R-))	359	(N? nil nil X?)	B
386.	(S () X? (- R-))	350	(M nil - X?)	B
387.	(S? () M (R- A+))	388	(M nil R- M)	B
388.	(S? () M (A+))	417	(X! A+ A+ M)	B
389.	(S? () M (R- A+ R-))	390	(M nil R- M)	B
390.	(S? () M (A+ R-))	418	(X! A+ A+ M)	B
391.	(S? () M?2 (R- R+ A-))	392	(M?2 nil R- M?2)	B
392.	(S? () M?2 (R+ A-))	407	(N?1 nil R+ M?2)	B
393.	(S? () M?2 (R- R+ A- R-))	394	(M?2 nil R- M?2)	B
394.	(S? () M?2 (R+ A- R-))	408	(N?1 nil R+ M?2)	B
395.	(S? () N? (R- - A+))	396	(N? nil R- N?)	B
396.	(S? () N? (- A+))	422	(X? nil nil N?)	B
397.	(S? () N? (R- - R- A+))	398	(N? nil R- N?)	B
398.	(S? () N? (- R- A+))	424	(X? nil nil N?)	B
399.	(S? () N? (R- - A+ R-))	400	(N? nil R- N?)	B
400.	(S? () N? (- A+ R-))	426	(X? nil nil N?)	B
401.	(S? () N? (R- - R- A+ R-))	402	(N? nil R- N?)	B
402.	(S? () N? (- R- A+ R-))	428	(X? nil nil N?)	B
403.	(S? (A+) N? ())	405	(N? R- nil N?)	B
404.	(S? (A+) N? (R-))	406	(N? R- nil N?)	B
405.	(S? (A+ R-) N? ())	431	(X? nil nil N?)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
406.	(S? (A+ R-) N? (R-))	405	(N? nil R- N?)	B
407.	(S? () N?1 (A-))	403	(N? A+ A- N?1)	B
408.	(S? () N?1 (A- R-))	404	(N? A+ A- N?1)	B
409.	(S? () S (+ A+))	388	(M nil + S)	B
410.	(S? () S (+ R- A+))	387	(M nil + S)	B
411.	(S? () S (+ A+ R-))	390	(M nil + S)	B
412.	(S? () S (+ R- A+ R-))	389	(M nil + S)	B
413.	(S? () SR (R- + A+))	409	(S nil R- SR)	B
414.	(S? () SR (R- + R- A+))	410	(S nil R- SR)	B
415.	(S? () SR (R- + A+ R-))	411	(S nil R- SR)	B
416.	(S? () SR (R- + R- A+ R-))	412	(S nil R- SR)	B
417.	(S? (A+) X! ())	431	(X? R- nil X!)	B
418.	(S? (A+) X! (R-))	417	(X! nil R- X!)	B
419.	(S? (A+ R-) X! ())	431	(X? R- nil X!)	B
420.	(S? (A+ R-) X! (R-))	419	(X! nil R- X!)	B
421.	(S? () X? (R- - A+))	395	(N? nil nil X?)	B
422.	(S? () X? (- A+))	388	(M nil - X?)	B
423.	(S? () X? (R- - R- A+))	397	(N? nil nil X?)	B
424.	(S? () X? (- R- A+))	387	(M nil - X?)	B
425.	(S? () X? (R- - A+ R-))	399	(N? nil nil X?)	B
426.	(S? () X? (- A+ R-))	390	(M nil - X?)	B
427.	(S? () X? (R- - R- A+ R-))	401	(N? nil nil X?)	B
428.	(S? () X? (- R- A+ R-))	389	(M nil - X?)	B
429.	(S? (A+) X? ())	486	(SR - A+ S?)	A
430.	(S? (A+) X? (R-))	404	(N? nil nil X?)	B
431.	(S? (A+ R-) X? ())	490	(SR - A+ S?)	A
432.	(S? (A+ R-) X? (R-))	406	(N? nil nil X?)	B
433.	(SR () M ())	435	(M R- nil M)	B
434.	(SR () M (R-))	436	(M R- nil M)	B
435.	(SR (R-) M ())	437	(M? + nil M)	B
436.	(SR (R-) M (R-))	438	(M? + nil M)	B
437.	(SR (R- +) M? ())	351	(S nil R- SR)	A
438.	(SR (R- +) M? (R-))	437	(M? nil R- M?)	B
439.	(SR (R- + A-) M?2 ())	353	(S nil R- SR)	A
440.	(SR (R- + A-) M?2 (R-))	439	(M?2 nil R- M?2)	B
441.	(SR (R- + A- R-) M?2 ())	355	(S nil R- SR)	A
442.	(SR (R- + A- R-) M?2 (R-))	441	(M?2 nil R- M?2)	B
443.	(SR () N? (R- -))	447	(N? R- nil N?)	B
444.	(SR () N? (-))	448	(N? R- nil N?)	B
445.	(SR () N? (R- - R-))	449	(N? R- nil N?)	B
446.	(SR () N? (- R-))	450	(N? R- nil N?)	B
447.	(SR (R-) N? (R- -))	448	(N? nil R- N?)	B
448.	(SR (R-) N? (-))	490	(X? nil nil N?)	B
449.	(SR (R-) N? (R- - R-))	450	(N? nil R- N?)	B
450.	(SR (R-) N? (- R-))	492	(X? nil nil N?)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
451.	(SR () S (+))	455	(S R- nil S)	B
452.	(SR () S (+ R-))	456	(S R- nil S)	B
453.	(SR (R- +) S ())	363	(S nil R- SR)	A
454.	(SR (R- + R-) S ())	364	(S nil R- SR)	A
455.	(SR (R-) S (+))	435	(M nil + S)	B
456.	(SR (R-) S (+ R-))	436	(M nil + S)	B
457.	(SR (R- + A+) S? ())	365	(S nil R- SR)	A
458.	(SR (R- + R- A+) S? ())	366	(S nil R- SR)	A
459.	(SR (R- + A+ R-) S? ())	367	(S nil R- SR)	A
460.	(SR (R- + R- A+ R-) S? ())	368	(S nil R- SR)	A
461.	(SR () SR (R- +))	467	(SR R- nil SR)	B
462.	(SR () SR (R- + R-))	468	(SR R- nil SR)	B
463.	(SR (R- +) SR ())	464	(SR R- nil SR)	A
464.	(SR (R- +) SR (R-))	453	(S nil R- SR)	B
465.	(SR (R- + R-) SR ())	466	(SR R- nil SR)	A
466.	(SR (R- + R-) SR (R-))	454	(S nil R- SR)	B
467.	(SR (R-) SR (R- +))	455	(S nil R- SR)	B
468.	(SR (R-) SR (R- + R-))	456	(S nil R- SR)	B
469.	(SR (R- + A+) SR? ())	470	(SR R- nil SR)	A
470.	(SR (R- + A+) SR? (R-))	457	(S? nil R- SR?)	B
471.	(SR (R- + R- A+) SR? ())	472	(SR R- nil SR)	A
472.	(SR (R- + R- A+) SR? (R-))	458	(S? nil R- SR?)	B
473.	(SR (R- + A+ R-) SR? ())	474	(SR R- nil SR)	A
474.	(SR (R- + A+ R-) SR? (R-))	459	(S? nil R- SR?)	B
475.	(SR (R- + R- A+ R-) SR? ())	476	(SR R- nil SR)	A
476.	(SR (R- + R- A+ R-) SR? (R-))	460	(S? nil R- SR?)	B
477.	(SR () X! (R- -))	478	(X! nil R- X!)	B
478.	(SR () X! (-))	490	(X? R- nil X!)	B
479.	(SR () X! (R- - R-))	480	(X! nil R- X!)	B
480.	(SR () X! (- R-))	492	(X? R- nil X!)	B
481.	(SR (R-) X! (R- -))	482	(X! nil R- X!)	B
482.	(SR (R-) X! (-))	490	(X? R- nil X!)	B
483.	(SR (R-) X! (R- - R-))	484	(X! nil R- X!)	B
484.	(SR (R-) X! (- R-))	492	(X? R- nil X!)	B
485.	(SR () X? (R- -))	443	(N? nil nil X?)	B
486.	(SR () X? (-))	433	(M nil - X?)	B
487.	(SR () X? (R- - R-))	445	(N? nil nil X?)	B
488.	(SR () X? (- R-))	434	(M nil - X?)	B
489.	(SR (R-) X? (R- -))	447	(N? nil nil X?)	B
490.	(SR (R-) X? (-))	435	(M nil - X?)	B
491.	(SR (R-) X? (R- - R-))	449	(N? nil nil X?)	B
492.	(SR (R-) X? (- R-))	436	(M nil - X?)	B
493.	(SR? () M (R- A+))	497	(M R- nil M)	B
494.	(SR? () M (A+))	498	(M R- nil M)	B
495.	(SR? () M (R- A+ R-))	499	(M R- nil M)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
496.	(SR? () M (A+ R-))	500	(M R- nil M)	B
497.	(SR? (R-) M (R- A+))	498	(M nil R- M)	B
498.	(SR? (R-) M (A+))	557	(X! A+ A+ M)	B
499.	(SR? (R-) M (R- A+ R-))	500	(M nil R- M)	B
500.	(SR? (R-) M (A+ R-))	558	(X! A+ A+ M)	B
501.	(SR? () M?2 (R- R+ A-))	505	(M?2 R- nil M?2)	B
502.	(SR? () M?2 (R+ A-))	506	(M?2 R- nil M?2)	B
503.	(SR? () M?2 (R- R+ A- R-))	507	(M?2 R- nil M?2)	B
504.	(SR? () M?2 (R+ A- R-))	508	(M?2 R- nil M?2)	B
505.	(SR? (R-) M?2 (R- R+ A-))	506	(M?2 nil R- M?2)	B
506.	(SR? (R-) M?2 (R+ A-))	531	(N?1 nil R+ M?2)	B
507.	(SR? (R-) M?2 (R- R+ A- R-))	508	(M?2 nil R- M?2)	B
508.	(SR? (R-) M?2 (R+ A- R-))	532	(N?1 nil R+ M?2)	B
509.	(SR? () N? (R- - A+))	521	(N? R- nil N?)	B
510.	(SR? () N? (- A+))	522	(N? R- nil N?)	B
511.	(SR? () N? (R- - R- A+))	523	(N? R- nil N?)	B
512.	(SR? () N? (- R- A+))	524	(N? R- nil N?)	B
513.	(SR? () N? (R- - A+ R-))	525	(N? R- nil N?)	B
514.	(SR? () N? (- A+ R-))	526	(N? R- nil N?)	B
515.	(SR? () N? (R- - R- A+ R-))	527	(N? R- nil N?)	B
516.	(SR? () N? (- R- A+ R-))	528	(N? R- nil N?)	B
517.	(SR? (R- A+) N? ())	519	(N? R- nil N?)	B
518.	(SR? (R- A+) N? (R-))	520	(N? R- nil N?)	B
519.	(SR? (R- A+ R-) N? ())	579	(X? nil nil N?)	B
520.	(SR? (R- A+ R-) N? (R-))	519	(N? nil R- N?)	B
521.	(SR? (R-) N? (R- - A+))	522	(N? nil R- N?)	B
522.	(SR? (R-) N? (- A+))	582	(X? nil nil N?)	B
523.	(SR? (R-) N? (R- - R- A+))	524	(N? nil R- N?)	B
524.	(SR? (R-) N? (- R- A+))	584	(X? nil nil N?)	B
525.	(SR? (R-) N? (R- - A+ R-))	526	(N? nil R- N?)	B
526.	(SR? (R-) N? (- A+ R-))	586	(X? nil nil N?)	B
527.	(SR? (R-) N? (R- - R- A+ R-))	528	(N? nil R- N?)	B
528.	(SR? (R-) N? (- R- A+ R-))	588	(X? nil nil N?)	B
529.	(SR? () N?1 (A-))	531	(N?1 R- nil N?1)	B
530.	(SR? () N?1 (A- R-))	532	(N?1 R- nil N?1)	B
531.	(SR? (R-) N?1 (A-))	517	(N? A+ A- N?1)	B
532.	(SR? (R-) N?1 (A- R-))	518	(N? A+ A- N?1)	B
533.	(SR? () S (+ A+))	537	(S R- nil S)	B
534.	(SR? () S (+ R- A+))	538	(S R- nil S)	B
535.	(SR? () S (+ A+ R-))	539	(S R- nil S)	B
536.	(SR? () S (+ R- A+ R-))	540	(S R- nil S)	B
537.	(SR? (R-) S (+ A+))	498	(M nil + S)	B
538.	(SR? (R-) S (+ R- A+))	497	(M nil + S)	B
539.	(SR? (R-) S (+ A+ R-))	500	(M nil + S)	B
540.	(SR? (R-) S (+ R- A+ R-))	499	(M nil + S)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
541.	(SR? () SR (R- + A+))	545	(SR R- nil SR)	B
542.	(SR? () SR (R- + R- A+))	546	(SR R- nil SR)	B
543.	(SR? () SR (R- + A+ R-))	547	(SR R- nil SR)	B
544.	(SR? () SR (R- + R- A+ R-))	548	(SR R- nil SR)	B
545.	(SR? (R-) SR (R- + A+))	537	(S nil R- SR)	B
546.	(SR? (R-) SR (R- + R- A+))	538	(S nil R- SR)	B
547.	(SR? (R-) SR (R- + A+ R-))	539	(S nil R- SR)	B
548.	(SR? (R-) SR (R- + R- A+ R-))	540	(S nil R- SR)	B
549.	(SR? () X! (R- - A+))	550	(X! nil R- X!)	B
550.	(SR? () X! (- A+))	582	(X? R- nil X!)	B
551.	(SR? () X! (R- - R- A+))	552	(X! nil R- X!)	B
552.	(SR? () X! (- R- A+))	584	(X? R- nil X!)	B
553.	(SR? () X! (R- - A+ R-))	554	(X! nil R- X!)	B
554.	(SR? () X! (- A+ R-))	586	(X? R- nil X!)	B
555.	(SR? () X! (R- - R- A+ R-))	556	(X! nil R- X!)	B
556.	(SR? () X! (- R- A+ R-))	588	(X? R- nil X!)	B
557.	(SR? (R- A+) X! ())	579	(X? R- nil X!)	B
558.	(SR? (R- A+) X! (R-))	557	(X! nil R- X!)	B
559.	(SR? (R- A+ R-) X! ())	579	(X? R- nil X!)	B
560.	(SR? (R- A+ R-) X! (R-))	559	(X! nil R- X!)	B
561.	(SR? (R-) X! (R- - A+))	562	(X! nil R- X!)	B
562.	(SR? (R-) X! (- A+))	582	(X? R- nil X!)	B
563.	(SR? (R-) X! (R- - R- A+))	564	(X! nil R- X!)	B
564.	(SR? (R-) X! (- R- A+))	584	(X? R- nil X!)	B
565.	(SR? (R-) X! (R- - A+ R-))	566	(X! nil R- X!)	B
566.	(SR? (R-) X! (- A+ R-))	586	(X? R- nil X!)	B
567.	(SR? (R-) X! (R- - R- A+ R-))	568	(X! nil R- X!)	B
568.	(SR? (R-) X! (- R- A+ R-))	588	(X? R- nil X!)	B
569.	(SR? () X? (R- - A+))	509	(N? nil nil X?)	B
570.	(SR? () X? (- A+))	494	(M nil - X?)	B
571.	(SR? () X? (R- - R- A+))	511	(N? nil nil X?)	B
572.	(SR? () X? (- R- A+))	493	(M nil - X?)	B
573.	(SR? () X? (R- - A+ R-))	513	(N? nil nil X?)	B
574.	(SR? () X? (- A+ R-))	496	(M nil - X?)	B
575.	(SR? () X? (R- - R- A+ R-))	515	(N? nil nil X?)	B
576.	(SR? () X? (- R- A+ R-))	495	(M nil - X?)	B
577.	(SR? (R- A+) X? ())	429	(S? nil R- SR?)	A
578.	(SR? (R- A+) X? (R-))	518	(N? nil nil X?)	B
579.	(SR? (R- A+ R-) X? ())	431	(S? nil R- SR?)	A
580.	(SR? (R- A+ R-) X? (R-))	520	(N? nil nil X?)	B
581.	(SR? (R-) X? (R- - A+))	521	(N? nil nil X?)	B
582.	(SR? (R-) X? (- A+))	498	(M nil - X?)	B
583.	(SR? (R-) X? (R- - R- A+))	523	(N? nil nil X?)	B
584.	(SR? (R-) X? (- R- A+))	497	(M nil - X?)	B
585.	(SR? (R-) X? (R- - A+ R-))	525	(N? nil nil X?)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
586.	(SR? (R-) X? (- A+ R-))	500	(M nil - X?)	B
587.	(SR? (R-) X? (R- - R- A+ R-))	527	(N? nil nil X?)	B
588.	(SR? (R-) X? (- R- A+ R-))	499	(M nil - X?)	B
589.	(X () M? (R- R+))	590	(M? nil R- M?)	B
590.	(X () M? (R+))	595	(N nil R+ M?)	B
591.	(X (A-) M?2 (R- R+))	592	(M?2 nil R- M?2)	B
592.	(X (A-) M?2 (R+))	596	(N?1 nil R+ M?2)	B
593.	(X (A- R-) M?2 (R- R+))	594	(M?2 nil R- M?2)	B
594.	(X (A- R-) M?2 (R+))	597	(N?1 nil R+ M?2)	B
595.	(X () N ())	598	(X nil nil N)	B
596.	(X (A-) N?1 ())	252	(N nil nil X)	A
597.	(X (A- R-) N?1 ())	253	(N nil nil X)	A
598.	(X () X ())	*Initial State*		
599.	(X! () S? (A+))	605	(SR - A+ S?)	B
600.	(X! () S? (A+ R-))	606	(SR - A+ S?)	B
601.	(X! (R-) S? (A+))	603	(SR - A+ S?)	B
602.	(X! (R-) S? (A+ R-))	604	(SR - A+ S?)	B
603.	(X! (R- -) SR ())	605	(X! nil R- X!)	A
604.	(X! (R- -) SR (R-))	606	(X! nil R- X!)	A
605.	(X! (-) SR ())	656	(X? R- nil X!)	A
606.	(X! (-) SR (R-))	656	(X? R- nil X!)	A
607.	(X! (R- - R-) SR ())	609	(X! nil R- X!)	A
608.	(X! (R- - R-) SR (R-))	610	(X! nil R- X!)	A
609.	(X! (- R-) SR ())	660	(X? R- nil X!)	A
610.	(X! (- R-) SR (R-))	660	(X? R- nil X!)	A
611.	(X! () SR? (R- A+))	599	(S? nil R- SR?)	B
612.	(X! () SR? (R- A+ R-))	600	(S? nil R- SR?)	B
613.	(X! (R- - A+) SR? ())	615	(X! nil R- X!)	A
614.	(X! (R- - A+) SR? (R-))	616	(X! nil R- X!)	A
615.	(X! (- A+) SR? ())	666	(X? R- nil X!)	A
616.	(X! (- A+) SR? (R-))	666	(X? R- nil X!)	A
617.	(X! (R- - R- A+) SR? ())	619	(X! nil R- X!)	A
618.	(X! (R- - R- A+) SR? (R-))	620	(X! nil R- X!)	A
619.	(X! (- R- A+) SR? ())	670	(X? R- nil X!)	A
620.	(X! (- R- A+) SR? (R-))	670	(X? R- nil X!)	A
621.	(X! (R- - A+ R-) SR? ())	623	(X! nil R- X!)	A
622.	(X! (R- - A+ R-) SR? (R-))	624	(X! nil R- X!)	A
623.	(X! (- A+ R-) SR? ())	674	(X? R- nil X!)	A
624.	(X! (- A+ R-) SR? (R-))	674	(X? R- nil X!)	A
625.	(X! (R- - R- A+ R-) SR? ())	627	(X! nil R- X!)	A
626.	(X! (R- - R- A+ R-) SR? (R-))	628	(X! nil R- X!)	A
627.	(X! (- R- A+ R-) SR? ())	678	(X? R- nil X!)	A
628.	(X! (- R- A+ R-) SR? (R-))	678	(X? R- nil X!)	A
629.	(X! (R-) SR? (R- A+))	601	(S? nil R- SR?)	B
630.	(X! (R-) SR? (R- A+ R-))	602	(S? nil R- SR?)	B

Membership Protocol Network States

Number	State	Previous	Transition Rule	Applied To
631.	(X? () M?2 (R- R+ A-))	632	(M?2 nil R- M?2)	B
632.	(X? () M?2 (R+ A-))	635	(N?1 nil R+ M?2)	B
633.	(X? (R-) M?2 (R- R+ A-))	634	(M?2 nil R- M?2)	B
634.	(X? (R-) M?2 (R+ A-))	636	(N?1 nil R+ M?2)	B
635.	(X? () N?1 (A-))	638	(S - A- N?1)	B
636.	(X? (R-) N?1 (A-))	637	(S - A- N?1)	B
637.	(X? (R- -) S ())	261	(N? nil nil X?)	A
638.	(X? (-) S ())	1	(M nil - X?)	A
639.	(X? (R- - R-) S ())	263	(N? nil nil X?)	A
640.	(X? (- R-) S ())	2	(M nil - X?)	A
641.	(X? () S? (A+))	655	(SR - A+ S?)	B
642.	(X? () S? (A+ R-))	656	(SR - A+ S?)	B
643.	(X? (R- - A+) S? ())	267	(N? nil nil X?)	A
644.	(X? (- A+) S? ())	4	(M nil - X?)	A
645.	(X? (R- - R- A+) S? ())	269	(N? nil nil X?)	A
646.	(X? (- R- A+) S? ())	3	(M nil - X?)	A
647.	(X? (R- - A+ R-) S? ())	271	(N? nil nil X?)	A
648.	(X? (- A+ R-) S? ())	6	(M nil - X?)	A
649.	(X? (R- - R- A+ R-) S? ())	273	(N? nil nil X?)	A
650.	(X? (- R- A+ R-) S? ())	5	(M nil - X?)	A
651.	(X? (R-) S? (A+))	653	(SR - A+ S?)	B
652.	(X? (R-) S? (A+ R-))	654	(SR - A+ S?)	B
653.	(X? (R- -) SR ())	277	(N? nil nil X?)	A
654.	(X? (R- -) SR (R-))	637	(S nil R- SR)	B
655.	(X? (-) SR ())	7	(M nil - X?)	A
656.	(X? (-) SR (R-))	638	(S nil R- SR)	B
657.	(X? (R- - R-) SR ())	281	(N? nil nil X?)	A
658.	(X? (R- - R-) SR (R-))	639	(S nil R- SR)	B
659.	(X? (- R-) SR ())	9	(M nil - X?)	A
660.	(X? (- R-) SR (R-))	640	(S nil R- SR)	B
661.	(X? () SR? (R- A+))	641	(S? nil R- SR?)	B
662.	(X? () SR? (R- A+ R-))	642	(S? nil R- SR?)	B
663.	(X? (R- - A+) SR? ())	287	(N? nil nil X?)	A
664.	(X? (R- - A+) SR? (R-))	643	(S? nil R- SR?)	B
665.	(X? (- A+) SR? ())	13	(M nil - X?)	A
666.	(X? (- A+) SR? (R-))	644	(S? nil R- SR?)	B
667.	(X? (R- - R- A+) SR? ())	291	(N? nil nil X?)	A
668.	(X? (R- - R- A+) SR? (R-))	645	(S? nil R- SR?)	B
669.	(X? (- R- A+) SR? ())	11	(M nil - X?)	A
670.	(X? (- R- A+) SR? (R-))	646	(S? nil R- SR?)	B
671.	(X? (R- - A+ R-) SR? ())	295	(N? nil nil X?)	A
672.	(X? (R- - A+ R-) SR? (R-))	647	(S? nil R- SR?)	B
673.	(X? (- A+ R-) SR? ())	17	(M nil - X?)	A
674.	(X? (- A+ R-) SR? (R-))	648	(S? nil R- SR?)	B
675.	(X? (R- - R- A+ R-) SR? ())	299	(N? nil nil X?)	A

Membership Protocol Network States

Number	*State*	*Previous*	*Transition Rule*	*Applied To*
676.	(X? (R- - R- A+ R-) SR? (R-))	649	(S? nil R- SR?)	B
677.	(X? (- R- A+ R-) SR? ())	15	(M nil - X?)	A
678.	(X? (- R- A+ R-) SR? (R-))	650	(S? nil R- SR?)	B
679.	(X? (R-) SR? (R- A+))	651	(S? nil R- SR?)	B
680.	(X? (R-) SR? (R- A+ R-))	652	(S? nil R- SR?)	B

The stable states in this enumeration are

(M () S ())	(M () SR ())	(X () X ())	(X () N ())
(S () M ())	(SR () M ())	(N () N ())	(N () X ())

Since the enumeration is exhaustive, all states reachable from these states have been included; thus adding any of these (or any of the others listed above) to our set of initial states would not have produced any additional output. Note that all stable network states we would desire (and none that we wouldn't) are represented, satisfying the consistency criterion for our protocol. Also, the closure requirement for our protocol is satisfied, since no processor ever received a message it did not have a state transition rule for.

Only after being assured of consistency and closure does it make sense to talk about other properties of the protocol, such as resistance to disconnection and resistance to forming cycles. These properties, unfortunately, bring into play other attributes of the network not directly modeled in our abstraction. As a result, it is not obvious how to devise a test similar to the above which will check for them. We can, however, recap our reasons for believing that our membership protocol actually exhibits these desired properties.

Resistance to disconnection depends on two properties of our protocol. The first, not directly expressed in the state transitions, is that only a leaf node may

attempt to remove itself from a tree. The second is the state transition sequence that a processor must follow in order to break a link. The processor breaking the link must have started out as master of the link; this guarantees that at most one of the processors at the ends of a link will be attempting to break the link at one time. When the attempt is made, the master will change to state X?. The considerations surrounding state X? which have already been discussed ensure that the link will only be broken if (1) the processor in state X? receives no references over that link during the period that the break is being confirmed, or (2) the processor in state X? rejoins the tree via some other neighbor before the break has been confirmed (this will cause a state transition to N? for the link being broken; this transition will force that link to be broken, preventing a cycle from forming).

Resistance to forming cycles is embodied in the property of states N and M? to turn away R+ messages seeking to establish new links. R+ messages are the only way new links can be formed, and only processors in state X (not currently part of the reference tree) will accept and act positively on them.

REFERENCES

1. Baker, H., "List Processing in Real Time on a Serial Computer," A.I. Working Paper 139, Artificial Intelligence Laboratory, M.I.T., February 1977.

2. Baker, H., and Hewitt, C., "The Incremental Garbage Collection of Processes," *ACM SIGART-SIGPLAN Symposium*, Rochester, N.Y., August 1977.

3. Barnes, G.H., et al., "The Illiac IV Computer," *IEEE Transactions C-17*, Vol. 8, August 1968.

4. Bishop, P., *Computer Systems with a Very Large Address Space and Garbage Collection*, LCS TR-178, Laboratory for Computer Science, M.I.T., May 1977.

5. Church, A., "The Calculi of Lambda Conversion," *Annals of Mathematics Studies*, Princeton University Press, 1941.

6. Curry, H.B., and Feys, R., *Combinatory Logic*, Amsterdam, 1958.

7. Dijkstra, E.W., "The Structure of the "THE" Multiprogramming System," *Communications of the ACM*, May 1968.

8. Farber, D.J., "A Ring Network," *Datamation*, February 1975.

9. Farber, D.J., and Heinrich, F.R., "The Structure of a Distributed Computer System: The Distributed File System," *Proceedings of the First International Conference on Computer Communications*, 1972.

10. Farber, D.J., et al., "The Distributed Computing System," *Proceedings of the Seventh Annual IEEE Computer Society International Conference*, February 1973.

11. Greif, I., *Semantics of Communicating Parallel Processes*, MAC TR-154, Project MAC, M.I.T., September 1975.

12. Gula, J., S.M. thesis, Department of Electrical Engineering and Computer Science, M.I.T., in preparation.

13. Henderson, D.A., *The Binding Model: A Semantic Base for Modular Programming Systems*, MAC TR-145, Project MAC, M.I.T., February 1975.

14. Hewitt, C., "Protection and Synchronization in Actor Systems," A.I. Working Paper 83, Artificial Intelligence Laboratory, M.I.T., November 1974.

15. Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," A.I. Working Paper 92, Artificial Intelligence Laboratory, M.I.T., April 1976.

16. Hewitt, C., and Baker, H., "Laws for Communicating Parallel Processes," A.I. Working Paper 134A, Artificial Intelligence Laboratory, M.I.T., May 1977.

17. Knuth, D., *Fundamental Algorithms*, Volume 1 of *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., February 1975, pp. 406-420.

18. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," Massachusetts Computer Associates Technical Report CA-7603-2911, March 1976.

19. Learning Research Group, *Personal Dynamic Media*, Xerox PARC Report SSL76-1, 1976.

20. Liskov, B., Snyder, A., Atkinson, R.,, and Schaffert, C., "Abstraction Mechanisms in CLU," Laboratory for Computer Science Computation Structures Group Memo 144-1, M.I.T., January 1977.

21. Metcalfe, R., *Packet Communication*, MAC TR-114, Project MAC, M.I.T., December 1973.

22. Metcalfe, R., and Boggs, D., *Ethernet: Distributed Packet Switching for Local Computer Networks*, Xerox PARC Report CSL75-7, November 1975.

23. Ornstein, S.M., et al., "Pluribus—A Reliable Multiprocessor," *Proceedings of the National Computer Conference*, May 1975.

24. Reed, D.P., and Kanodia, R.K., *Eventcounts: A New Model for Process Synchronization*, Project MAC Computer Systems Research Division RFC#102, M.I.T., January 1976.

25. Rowe, L.A., "The Distributed Computing Operating System," Department of Information and Computer Science Technical Report 66, University of California at Irvine, June 1975.

26. Rowe, L.A., Hopwood, M.D., and Farber, D.J., "Software Methods for Achieving Fail-Soft Behavior in the Distributed Computing System," *Proceedings of the IEEE Symposium on Computer Software Reliability*, 1973.

27. Rumbaugh, J., *A Parallel Asynchronous Computer Architecture for Data Flow Programs*, MAC TR-150, Project MAC, M.I.T., May 1975.

28. Stearns, R.E., Lewis, P.M., and Rosenkrantz, D.J., "Concurrency Control for Database Systems," *IEEE Symposium on Foundations of Computer Science CH1133-8C*, October 1976.

29. Steele, G., "LAMBDA: The Ultimate Declarative," A.I. Memo 379, Artificial Intelligence Laboratory, M.I.T., November 1976.

30. Steele, G., and Sussman, G., "LAMBDA: The Ultimate Imperative," A.I. Memo 353, Artificial Intelligence Laboratory, M.I.T., March 1976.

31. Steiger, R., *Actor Machine Architecture*, S.M. Thesis, Department of Electrical Engineering, M.I.T., May 1974.

32. Strachey, C., and Wadsworth, C.P., "Continuations: A Mathematical Semantics for Handling Full Jumps," Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

33. Sussman, G., and Steele, G., "SCHEME: An Interpreter for Extended Lambda Calculus," A.I. Memo 349, Artificial Intelligence Laboratory, M.I.T., December 1975.

34. Ward, S., *Functional Domains of Applicative Languages*, MAC TR-136, Project MAC, M.I.T., September 1974.

35. Ward, S., and Halstead, R., "A Syntactic Theory of Message Passing," internal memorandum, M.I.T., September 1976.

36. Wulf, W., and Levin, R., "C.mmp—A Multi-Mini-Processor," *AFIPS Conference Proceedings*, Fall 1972.

37. Wulf, W., et al., "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, June 1974.

Official Distribution List

Defense Documentation Center
Cameron Station
Alexandria, Va 22314 — 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, Va 22217 — 2 copies

Office of Naval Research
Code 102IP
Arlington, Va 22217 — 6 copies

Office of Naval Research
Code 200
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 455
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 458
Arlington, Va 22217 — 1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, Ma 02210 — 1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Il 60605 — 1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, Ca 91106 — 1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003 — 1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375 — 6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380 — 1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, Ca 92152 — 1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Mathematics Department
Bethesda, Md 20084 — 1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy